



Astropy Documentation

Release 0.4.2

The Astropy Developers

November 13, 2014

I	User Documentation	3
1	What's New in Astropy 0.4?	5
1.1	Overview	5
1.2	Coordinates	5
1.3	SAMP	6
1.4	Quantity	6
1.5	Inspecting FITS headers from the command line	7
1.6	Reading and writing HTML tables	7
1.7	Documentation URL changes	7
1.8	astropy-helpers package	7
1.9	Configuration	7
1.10	Deprecation and backward-incompatible changes	7
1.11	Full change log	8
1.12	Note on future versions	8
2	Overview	9
2.1	Astropy Project Concept	9
2.2	astropy Core Package	9
2.3	Affiliated Packages	9
2.4	Community	10
3	Installation	11
3.1	Requirements	11
3.2	Installing Astropy	11
3.3	Building from source	12
4	Getting Started with Astropy	17
4.1	Importing Astropy	17
4.2	Getting started with subpackages	18
5	Constants (astropy.constants)	19
5.1	Introduction	19
5.2	Getting Started	19
5.3	Reference/API	20
6	Units and Quantities (astropy.units)	23
6.1	Introduction	23
6.2	Getting Started	23
6.3	Using astropy.units	25
6.4	See Also	43
6.5	Reference/API	43

6.6	Acknowledgments	99
7	N-dimensional datasets (<code>astropy.nddata</code>)	101
7.1	Introduction	101
7.2	Getting started	101
7.3	Using <code>nddata</code>	102
7.4	Reference/API	105
8	Data Tables (<code>astropy.table</code>)	117
8.1	Introduction	117
8.2	Getting Started	117
8.3	Using <code>table</code>	120
8.4	Reference/API	156
9	Time and Dates (<code>astropy.time</code>)	189
9.1	Introduction	189
9.2	Getting Started	189
9.3	Using <code>astropy.time</code>	190
9.4	Reference/API	201
9.5	Acknowledgments and Licenses	222
10	Astronomical Coordinate Systems (<code>astropy.coordinates</code>)	223
10.1	Introduction	223
10.2	Getting Started	223
10.3	Overview of <code>astropy.coordinates</code> concepts	226
10.4	Using <code>astropy.coordinates</code>	226
10.5	Migrating from pre-v0.4 coordinates	255
10.6	See Also	255
10.7	Reference/API	256
11	World Coordinate System (<code>astropy.wcs</code>)	309
11.1	Introduction	309
11.2	Getting Started	309
11.3	Using <code>astropy.wcs</code>	310
11.4	Supported projections	312
11.5	Other information	313
11.6	See Also	319
11.7	Reference/API	319
11.8	Acknowledgments and Licenses	367
12	Models and Fitting (<code>astropy.modeling</code>)	369
12.1	Introduction	369
12.2	Getting started	369
12.3	Using <code>astropy.modeling</code>	372
12.4	Reference/API	389
13	Unified file read/write interface	487
13.1	Getting started with Table I/O	487
13.2	Built-in table readers/writers	487
14	FITS File handling (<code>astropy.io.fits</code>)	493
14.1	Introduction	493
14.2	Getting Started	493
14.3	Using <code>astropy.io.fits</code>	502
14.4	Other Information	532

14.5	Reference/API	593
15	ASCII Tables (<code>astropy.io.ascii</code>)	683
15.1	Introduction	683
15.2	Getting Started	683
15.3	Supported formats	685
15.4	Using <code>astropy.io.ascii</code>	685
15.5	Reference/API	701
16	VOTable XML handling (<code>astropy.io.votable</code>)	733
16.1	Introduction	733
16.2	Getting Started	733
16.3	Using <code>astropy.io.votable</code>	735
16.4	See Also	738
16.5	Reference/API	738
17	Miscellaneous Input/Output (<code>astropy.io.misc</code>)	781
17.1	<code>astropy.io.misc</code> Module	781
17.2	<code>astropy.io.misc.hdf5</code> Module	782
18	Convolution and filtering (<code>astropy.convolution</code>)	785
18.1	Introduction	785
18.2	Getting started	785
18.3	Using <code>astropy.convolution</code>	787
18.4	Reference/API	792
19	Cosmological Calculations (<code>astropy.cosmology</code>)	815
19.1	Introduction	815
19.2	Getting Started	815
19.3	Using <code>astropy.cosmology</code>	816
19.4	For Developers: Using <code>astropy.cosmology</code> inside Astropy	819
19.5	See Also	820
19.6	Range of validity and reliability	820
19.7	Reference/API	820
20	Astrostatistics Tools (<code>astropy.stats</code>)	855
20.1	Introduction	855
20.2	Getting Started	855
20.3	See Also	855
20.4	Reference/API	855
21	Virtual Observatory Access (<code>astropy.vo</code>)	867
21.1	Introduction	867
22	Configuration system (<code>astropy.config</code>)	945
22.1	Introduction	945
22.2	Getting Started	945
22.3	Using <code>astropy.config</code>	946
22.4	Adding new configuration items	947
22.5	See Also	950
22.6	Reference/API	953
23	I/O Registry (<code>astropy.io.registry</code>)	959
23.1	Introduction	959
23.2	Using <code>astropy.io.registry</code>	959

23.3	Reference/API	960
24	Logging system	965
24.1	Overview	965
24.2	Configuring the logging system	965
24.3	Context managers	966
24.4	Using the configuration file	966
24.5	Reference/API	967
25	Python warnings system	973
26	Astropy Core Package Utilities (<code>astropy.utils</code>)	975
26.1	Introduction	975
26.2	Reference/API	975
27	Current status of sub-packages	1015
28	Major Release History	1017
28.1	What's New in Astropy 0.4?	1017
28.2	What's New in Astropy 0.3?	1020
28.3	What's New in Astropy 0.2	1020
28.4	What's New in Astropy 0.1	1020
29	Known Issues	1021
29.1	Quantities lose their units with some operations	1021
29.2	Some docstrings can not be displayed in IPython < 0.13.2	1021
29.3	Locale errors	1022
29.4	Floating point precision issues on Python 2.6 on Microsoft Windows	1023
29.5	Failing logging tests when running the tests in IPython	1023
29.6	mmap support for <code>astropy.io.fits</code> on GNU Hurd	1023
29.7	Crash on upgrading from Astropy 0.2 to a newer version	1023
29.8	Color printing on Windows	1023
29.9	Table sorting can silently fail on MacOS X or Windows with Python 3 and Numpy < 1.6.2	1023
29.10	Anaconda users should upgrade with <code>conda</code> , not <code>pip</code>	1024
29.11	Installation fails on Mageia-2 or Mageia-3 distributions	1024
29.12	Remote data utilities in <code>astropy.utils.data</code> fail on some Python distributions	1024
29.13	Very long integers in ASCII tables silently converted to float for Numpy 1.5	1024
30	Authors and Credits	1027
30.1	Astropy Project Coordinators	1027
30.2	Core Package Contributors	1027
30.3	Other Credits	1030
31	Licenses	1031
31.1	Astropy License	1031
31.2	Other Licenses	1031
II	Getting help	1033
III	Reporting Issues	1037
32	For astropy-helpers	1041

IV	Contributing	1043
33	Try the development version	1047
33.1	Overview	1047
33.2	Step-by-step instructions	1048
34	How to make a code contribution	1053
34.1	Pre-requisites	1053
34.2	Strongly Recommended, but not required	1053
34.3	New to git?	1053
34.4	Astropy Guidelines for git	1055
34.5	Workflow	1055
34.6	Fetch the latest Astropy	1055
34.7	Make a new feature branch	1056
34.8	Install your branch	1056
34.9	The editing workflow	1057
34.10	Add a changelog entry	1058
34.11	Copy your changes to GitHub	1058
34.12	Ask for your changes to be reviewed	1058
34.13	Revise and push as necessary	1059
34.14	Rebase, but only if asked	1059
V	Developer Documentation	1061
35	How to make a code contribution	1065
35.1	Pre-requisites	1065
35.2	Strongly Recommended, but not required	1065
35.3	New to git?	1065
35.4	Astropy Guidelines for git	1067
35.5	Workflow	1067
35.6	Fetch the latest Astropy	1067
35.7	Make a new feature branch	1068
35.8	Install your branch	1068
35.9	The editing workflow	1069
35.10	Add a changelog entry	1070
35.11	Copy your changes to GitHub	1070
35.12	Ask for your changes to be reviewed	1070
35.13	Revise and push as necessary	1071
35.14	Rebase, but only if asked	1071
36	Coding Guidelines	1073
36.1	Interface and Dependencies	1073
36.2	Documentation and Testing	1074
36.3	Data and Configuration	1074
36.4	Standard output, warnings, and errors	1075
36.5	Coding Style/Conventions	1075
36.6	Unicode guidelines	1076
36.7	Including C Code	1077
36.8	Writing portable code for Python 2 and 3	1078
36.9	Requirements Specific to Affiliated Packages	1080
36.10	Examples	1080
36.11	Additional Resources	1084
37	Writing Documentation	1087

37.1	Building the Documentation from source	1087
37.2	Astropy Documentation Rules and Guidelines	1087
37.3	Sphinx Documentation Themes	1095
37.4	Sphinx extensions	1095
38	Testing Guidelines	1099
38.1	Testing Framework	1099
38.2	Running Tests	1099
38.3	Writing tests	1102
38.4	Writing doctests	1109
39	Writing Command-Line Scripts	1113
39.1	Example	1113
40	Building Astropy and its Subpackages	1115
40.1	Astropy-helpers	1115
40.2	Customizing setup/build for subpackages	1115
41	C or Cython Extensions	1117
41.1	Installing C header files	1117
41.2	Preventing importing at build time	1117
42	Release Procedures	1119
42.1	Release Procedure	1119
42.2	Maintaining Bug Fix Releases	1122
42.3	Creating a GPG Signing Key and a Signed Tag	1125
42.4	Creating a MacOS X Installer on a DMG	1126
43	Workflow for Maintainers	1129
43.1	Integrating changes via the web interface (recommended)	1129
43.2	Integrating changes manually	1129
43.3	Using Milestones and Labels	1130
43.4	Updating and Maintaining the Changelog	1131
44	How to create and maintain an Astropy affiliated package	1133
44.1	Starting a new package	1133
44.2	Releasing an affiliated package	1136
45	Full Changelog	1139
45.1	0.4.2 (2014-09-23)	1139
45.2	0.4.1 (2014-08-08)	1140
45.3	0.4 (2014-07-16)	1142
45.4	0.3.2 (2014-05-13)	1151
45.5	0.3.1 (2014-03-04)	1153
45.6	0.3 (2013-11-20)	1156
45.7	0.2.5 (2013-10-25)	1166
45.8	0.2.4 (2013-07-24)	1168
45.9	0.2.3 (2013-05-30)	1169
45.10	0.2.2 (2013-05-21)	1169
45.11	0.2.1 (2013-04-03)	1171
45.12	0.2 (2013-02-19)	1173
45.13	0.1 (2012-06-19)	1177

VI Indices and Tables	1179
Bibliography	1183
Python Module Index	1185
Index	1187



Welcome to the Astropy documentation! Astropy is a community-driven package intended to contain much of the core functionality and some common tools needed for performing astronomy and astrophysics with Python.

Part I

User Documentation

WHAT'S NEW IN ASTROPY 0.4?

1.1 Overview

Astropy 0.4 is a major release that adds new functionality since the 0.3.x series of releases. A new sub-package is included (see [SAMP](#)), a major overhaul of the *Coordinates* sub-package has been completed (see [Coordinates](#)), and many new features and improvements have been implemented for the existing sub-packages. In addition to usability improvements, we have made a number of changes in the infrastructure for setting up/installing the package (see [astropy-helpers package](#)), as well as reworking the configuration system (see [Configuration](#)).

In addition to these major changes, a large number of smaller improvements have occurred. Since v0.3, by the numbers:

- 819 issues have been closed
- 511 pull requests have been merged
- 57 distinct people have contributed code

1.2 Coordinates

The *Astronomical Coordinate Systems* (*astropy.coordinates*) sub-package has been largely re-designed based on broad community discussion and experience with v0.2 and v0.3. The key motivation was to implement coordinates within an extensible framework that cleanly separates the distinct aspects of data representation, coordinate frame representation and transformation, and user interface. This is described in the [APE5](#) document. Details of the new usage are given in the *Astronomical Coordinate Systems* (*astropy.coordinates*) section of the documentation.

An important point is that this sub-package is now considered stable and we do not expect any further major interface changes.

For most users the major change is that the recommended user interface to coordinate functionality is the *SkyCoord* class instead of classes like *ICRS* or *Galactic* (which are now called “frame” classes). For example:

```
>>> from astropy import units as u
>>> from astropy.coordinates import SkyCoord
>>> coordinate = SkyCoord(123.4*u.deg, 56.7*u.deg, frame='icrs')
```

The frame classes can still be used to create coordinate objects as before, but they are now more powerful because they can represent abstract coordinate frames without underlying data. The more typical use for frame classes is now:

```
>>> from astropy.coordinates import FK4 # Or ICRS, Galactic, or similar
>>> fk4_frame = FK4(equinox='J1980.0', obstime='2011-06-12T01:12:34')
>>> coordinate.transform_to(fk4_frame)
<SkyCoord (FK4): equinox=J1980.000, obstime=2011-06-12T01:12:34.000, ra=123.001698182 deg, dec=56.760000000000004 deg>
```

At the lowest level of the framework are the representation classes which describe how to represent a point in a frame as a tuple of quantities, for instance as spherical, cylindrical, or cartesian coordinates. Any coordinate object can now be created using values in a number of common representations and be displayed using those representations. For example:

```
>>> coordinate = SkyCoord(1*u.pc, 2*u.pc, 3*u.pc, representation='cartesian')
>>> coordinate
<SkyCoord (ICRS): x=1.0 pc, y=2.0 pc, z=3.0 pc>

>>> coordinate.representation = 'physicsspherical'
>>> coordinate
<SkyCoord (ICRS): phi=63.4349488229 deg, theta=36.6992252005 deg, r=3.74165738677 pc>
```

1.3 SAMP

The *SAMP* (*Simple Application Messaging Protocol* (*astropy.vo.samp*) sub-package is a new sub-package (adapted from the *SAMPy* package) that contains an implementation of the Simple Application Messaging Protocol (SAMP) standard that allows communication with any SAMP-enabled application (such as *TOPCAT*, *SAO Ds9*, and *Aladin*). This sub-package includes both classes for a hub and a client, as well as an *integrated client* which automatically connects to any running SAMP hub and acts as a client:

```
>>> from astropy.vo.samp import SAMPIntegratedClient
>>> client = SAMPIntegratedClient()
>>> client.connect()
```

We can then use the client to communicate with other clients:

```
>>> client.get_registered_clients()
['hub', 'c1', 'c2']
>>> client.get_metadata('c1')
{'author.affiliation': 'Astrophysics Group, Bristol University',
 'author.email': 'm.b.taylor@bristol.ac.uk',
 'author.name': 'Mark Taylor',
 'home.page': 'http://www.starlink.ac.uk/topcat/',
 'samp.description.text': 'Tool for OPerations on Catalogues And Tables',
 'samp.documentation.url': 'http://127.0.0.1:2525/doc/sun253/index.html',
 'samp.icon.url': 'http://127.0.0.1:2525/doc/images/tc_sok.gif',
 'samp.name': 'topcat',
 'topcat.version': '4.0-1'}
```

and we can then send for example tables and images over SAMP to other applications (see *SAMP (Simple Application Messaging Protocol)* (*astropy.vo.samp*) for examples of how to do this).

1.4 Quantity

The *Quantity* class has seen a series of optimizations and is now substantially faster. Additionally, the *time*, *coordinates*, and *table* subpackages integrate better with *Quantity*, with further improvements on the way for *table*. See *Quantity* and the other subpackage documentation sections for more details.

1.5 Inspecting FITS headers from the command line

The *FITS File handling* (*astropy.io.fits*) sub-package now provides a command line script for inspecting the header(s) of a FITS file. With Astropy 0.4 installed, run `fitsheader file.fits` in your terminal to print the header information to the screen in a human-readable format. Run `fitsheader --help` to see the full usage documentation.

1.6 Reading and writing HTML tables

The *ASCII Tables* (*astropy.io.ascii*) sub-package now provides the capability to read a table within an HTML file or web URL into an `astropy Table` object. This requires the `BeautifulSoup4` package to be installed. Conversely a `Table` object can now be written out as an HTML table.

1.7 Documentation URL changes

Starting in v0.4, the astropy documentation (and any package that uses `astropy-helpers`) will show the full name of functions and classes prefixed by the intended user-facing location. This is in contrast to previous versions, which pointed to the actual implementation module, rather than the intended public API location.

This will affect URLs pointing to specific documentation pages. For example, this URL points to the v0.3 location of the `astropy.cosmology.luminosity_distance` function:

- http://docs.astropy.org/en/v0.3/api/astropy.cosmology.funcs.luminosity_distance.html

while the appropriate URL for v0.4 and later is:

- http://docs.astropy.org/en/v0.4/api/astropy.cosmology.luminosity_distance.html

1.8 astropy-helpers package

We have now extracted our set-up and documentation utilities into a separate package, `astropy-helpers`. In practice, this does not change anything from a user point of view, but it is a big internal change that will allow any other packages to benefit from the set-up utilities developed for the core package without having to first install astropy.

1.9 Configuration

The configuration framework has been re-factored based on the design described in `APE3`. If you have previously edited the astropy configuration file (typically located at `~/.astropy/config/astropy.cfg`) then you should read over *Configuration transition* in order to understand how to update it to the new mechanism.

1.10 Deprecation and backward-incompatible changes

- Quantity comparisons with `==` or `!=` now always return `True` or `False`, even if units do not match (for which case a `UnitsError` used to be raised). [#2328]
- The functional interface for `astropy.cosmology` (e.g. `cosmology.H(z=0.5)`) is now deprecated in favor of the objected-oriented approach (`WMAP9.H(z=0.5)`). [#2343]

- The `astropy.coordinates` sub-package has undergone major changes for implementing the [APE5](#) plan for the package. A compatibility layer has been added that will allow common use cases of pre-v0.4 coordinates to work, but this layer will be removed in the next major version. Hence, any use of the coordinates package should be adapted to the new framework. Additionally, the compatibility layer cannot be used for convenience functions (like the `match_catalog_*` functions), as these have been moved to `SkyCoord`. From this point on, major changes to the coordinates classes are not expected. [#2422]
- The configuration framework has been re-designed to the scheme of [APE3](#). The previous framework based on `ConfigurationItem` is deprecated, and will be removed in a future release. Affiliated packages should update to the new configuration system, and any users who have customized their configuration file should migrate to the new configuration approach. Until they do, warnings will appear prompting them to do so.

1.11 Full change log

To see a detailed list of all changes in version 0.4 and prior, please see the [Full Changelog](#).

1.12 Note on future versions

While the current release supports Python 2.6, 2.7, and 3.1 to 3.4, the next release (1.0) will drop support for Python 3.1 and 3.2.

Astropy at a glance

OVERVIEW

Here we describe a broad overview of the Astropy project and its parts.

2.1 Astropy Project Concept

The “Astropy Project” is distinct from the `astropy` package. The Astropy Project is a process intended to facilitate communication and interoperability of python packages/codes in astronomy and astrophysics. The project thus encompasses the `astropy` core package (which provides a common framework), all “affiliated packages” (described below in [Affiliated Packages](#)), and a general community aimed at bringing resources together and not duplicating efforts.

2.2 `astropy` Core Package

The `astropy` package (alternatively known as the “core” package) contains various classes, utilities, and a packaging framework intended to provide commonly-used astronomy tools. It is divided into a variety of sub-packages, which are documented in the remainder of this documentation (see [User Documentation](#) for documentation of these components).

The core also provides this documentation, and a variety of utilities that simplify starting other python astronomy/astrophysics packages. As described in the following section, these simplify the process of creating affiliated packages.

2.3 Affiliated Packages

The Astropy project includes the concept of “affiliated packages.” An affiliated package is an astronomy-related python package that is not part of the `astropy` core source code, but has requested to be included in the general community effort of the Astropy project. Such a package may be a candidate for eventual inclusion in the main `astropy` package (although this is not required). Until then, however, it is a separate package, and may not be in the `astropy` namespace.

The authoritative list of current affiliated packages is available at <http://affiliated.astropy.org>, including a machine-readable [JSON file](#).

If you are interested in starting an affiliated package, or have a package you are interested in making more compatible with `astropy`, the `astropy` core package includes features that simplify and homogenize package management. Astropy provides a [package template](#) that provides a common way to organize a package, to make your life simpler. You can use this template either with a new package you are starting or an existing package to give it most of the organizational tools Astropy provides, including the documentation, testing, and Cython-building tools. See the [usage instructions in the template](#) for further details.

To then get your package listed on the registry, take a look at the [guidelines for becoming an affiliated package](#), and then post your intent on the [astropy-dev mailing list](#). The Astropy coordination committee, in consultation with the community, will provide you feedback on the package, and will add it to the registry when it is approved.

2.4 Community

Aside from the actual code, Astropy is also a community of astronomy- associated users and developers that agree that sharing utilities is healthy for the community and the science it produces. This community is of course central to accomplishing anything with the code itself. We follow the [Python Software Foundation Code of Conduct](#) and welcome anyone who wishes to contribute to the project.

INSTALLATION

3.1 Requirements

Astropy has the following strict requirements:

- Python 2.6 ($\geq 2.6.5$), 2.7, 3.1, 3.2, 3.3, or 3.4
- Numpy 1.5.1 or later

Astropy also depends on other packages for optional features:

- `h5py`: To read/write `Table` objects from/to HDF5 files
- `BeautifulSoup`: To read `Table` objects from HTML files
- `scipy`: To power a variety of features (currently mainly cosmology-related functionality)
- `xmllint`: To validate VOTABLE XML files.

However, note that these only need to be installed if those particular features are needed. Astropy will import even if these dependencies are not installed.

3.2 Installing Astropy

3.2.1 Using pip

To install Astropy with `pip`, simply run:

```
pip install --no-deps astropy
```

Warning: Users of the Anaconda python distribution should follow the instructions for *Anaconda python distribution*.

Note: You will need a C compiler (e.g. `gcc` or `clang`) to be installed (see [Building from source](#) below) for the installation to succeed.

Note: The `--no-deps` flag is optional, but highly recommended if you already have Numpy installed, since otherwise `pip` will sometimes try to “help” you by upgrading your Numpy installation, which may not always be desired.

Note: If you get a `PermissionError` this means that you do not have the required administrative access to

install new packages to your Python installation. In this case you may consider using the `--user` option to install the package into your home directory. You can read more about how to do this in the [pip documentation](#).

Alternatively, if you intend to do development on other software that uses Astropy, such as an affiliated package, consider installing Astropy into a *virtualenv*.

Do **not** install Astropy or other third-party packages using `sudo` unless you are fully aware of the risks.

3.2.2 Anaconda python distribution

Astropy is installed by default with Anaconda. To update to the latest version run:

```
conda update astropy
```

Note: There may be a delay of a day or to between when a new version of Astropy is released and when a package is available for Anaconda. You can check for the list of available versions with `conda search astropy`.

Note: Attempting to use `pip` to upgrade your installation of Astropy may result in a corrupted installation.

3.2.3 Binary installers

Binary installers are available on Windows for Python 2.6, 2.7, 3.1, and 3.2 at [PyPI](#).

3.2.4 Testing an installed Astropy

The easiest way to test your installed version of astropy is running correctly is to use the `astropy.test()` function:

```
import astropy
astropy.test()
```

The tests should run and print out any failures, which you can report at the [Astropy issue tracker](#).

Note: This way of running the tests may not work if you do it in the astropy source distribution. See [Testing a source code build of Astropy](#) for how to run the tests from the source code directory, or [Running Tests](#) for more details.

Note: Running the tests this way is currently disabled in the IPython REPL due to conflicts with some common display settings in IPython. Please run the Astropy tests under the standard Python command-line interpreter.

3.3 Building from source

3.3.1 Prerequisites

You will need a compiler suite and the development headers for Python and Numpy in order to build Astropy. On Linux, using the package manager for your distribution will usually be the easiest route, while on MacOS X you will need the XCode command line tools.

The [instructions for building Numpy from source](#) are also a good resource for setting up your environment to build Python packages.

You will also need [Cython](#) (v0.15 or later) installed to build from source, unless you are installing a numbered release. (The releases packages have the necessary C files packaged with them, and hence do not require Cython.)

Note: If you are using MacOS X, you will need to the XCode command line tools. One way to get them is to install [XCode](#). If you are using OS X 10.7 (Lion) or later, you must also explicitly install the command line tools. You can do this by opening the XCode application, going to **Preferences**, then **Downloads**, and then under **Components**, click on the Install button to the right of **Command Line Tools**. Alternatively, on 10.7 (Lion) or later, you do not need to install XCode, you can download just the command line tools from <https://developer.apple.com/downloads/index.action> (requires an Apple developer account).

3.3.2 Obtaining the source packages

Source packages

The latest stable source package for Astropy can be [downloaded here](#).

Development repository

The latest development version of Astropy can be cloned from github using this command:

```
git clone git://github.com/astropy/astropy.git
```

Note: If you wish to participate in the development of Astropy, see [Developer Documentation](#). This document covers only the basics necessary to install Astropy.

3.3.3 Building and Installing

Astropy uses the Python [distutils](#) framework for building and installing and requires the [distribute](#) extension—the later is automatically downloaded when running `python setup.py` if it is not already provided by your system.

If Numpy is not already installed in your Python environment, the astropy setup process will try to download and install it before continuing to install astropy.

To build Astropy (from the root of the source tree):

```
python setup.py build
```

To install Astropy (from the root of the source tree):

```
python setup.py install
```

3.3.4 Troubleshooting

If you get an error mentioning that you do not have the correct permissions to install Astropy into the default `site-packages` directory, you can try installing with:

```
python setup.py install --user
```

which will install into a default directory in your home directory.

External C libraries

The Astropy source ships with the C source code of a number of libraries. By default, these internal copies are used to build Astropy. However, if you wish to use the system-wide installation of one of those libraries, you can pass one or more of the `--use-system-X` flags to the `setup.py` build command.

For example, to build Astropy using the system `libexpat`, use:

```
python setup.py build --use-system-expat
```

To build using all of the system libraries, use:

```
python setup.py build --use-system-libraries
```

To see which system libraries Astropy knows how to build against, use:

```
python setup.py build --help
```

As with all distutils commandline options, they may also be provided in a `setup.cfg` in the same directory as `setup.py`. For example, to use the system `libexpat`, add the following to the `setup.cfg` file:

```
[build]
use_system_expat=1
```

The required version of setuptools is not available

If upon running the `setup.py` script you get a message like

```
The required version of setuptools (>=0.9.8) is not available, and can't be installed while this script is running. Please install a more recent version first, using 'easy_install -U setuptools'.
```

```
(Currently using setuptools 0.6c11 (/path/to/setuptools-0.6c11-py2.7.egg))
```

this is because you have a very outdated version of the `setuptools` package which is used to install Python packages. Normally Astropy will bootstrap newer version of `setuptools` via the network, but `setuptools` suggests that you first *uninstall* the old version (the `easy_install -U setuptools` command).

However, in the likely case that your version of `setuptools` was installed by an OS system package (on Linux check your package manager like `apt` or `yum` for a package called `python-setuptools`), trying to uninstall with `easy_install` and without using `sudo` may not work, or may leave your system package in an inconsistent state.

As the best course of action at this point depends largely on the individual system and how it is configured, if you are not sure yourself what do please ask on the Astropy mailing list.

The Windows installer can't find Python in the registry

This is a common issue with Windows installers for Python packages that do not support the new User Access Control (UAC) framework added in Windows Vista and later. In particular, when a Python is installed “for all users” (as opposed to for a single user) it adds entries for that Python installation under the `HKEY_LOCAL_MACHINE` (HKLM) hierarchy and *not* under the `HKEY_CURRENT_USER` (HKCU) hierarchy. However, depending on your UAC settings, if the Astropy installer is not executed with elevated privileges it will not be able to check in HKLM for the required information about your Python installation.

In short: If you encounter this problem it's because you need the appropriate entries in the Windows registry for Python. You can download [this script](#) and execute it with the same Python as the one you want to install Astropy into. For example to add the missing registry entries to your Python 2.7:

```
C:\>C:\Python27\python.exe C:\Path\To\Downloads\win_register_python.py
```

3.3.5 Building documentation

Note: Building the documentation is in general not necessary unless you are writing new documentation or do not have internet access, because the latest (and archive) versions of astropy's documentation should be available at docs.astropy.org.

Building the documentation requires the Astropy source code and some additional packages:

- [Sphinx](#) (and its dependencies) 1.0 or later
- [Graphviz](#)
- [Astropy-helpers](#) (Astropy and most affiliated packages include this as a submodule in the source repository, so it does not need to be installed separately.)

Note: Sphinx also requires a reasonably modern LaTeX installation to render equations. Per the [Sphinx documentation](#), for the TexLive distribution the following packages are required to be installed:

- latex-recommended
- latex-extra
- fonts-recommended

For other LaTeX distributions your mileage may vary. To build the PDF documentation using LaTeX, the `fonts-extra` TexLive package or the `inconsolata` CTAN package are also required.

There are two ways to build the Astropy documentation. The most straightforward way is to execute the command (from the astropy source directory):

```
python setup.py build_sphinx
```

The documentation will be built in the `docs/_build/html` directory, and can be read by pointing a web browser to `docs/_build/html/index.html`.

The LaTeX documentation can be generated by using the command:

```
python setup.py build_sphinx -b latex
```

The LaTeX file `Astropy.tex` will be created in the `docs/_build/latex` directory, and can be compiled using `pdflatex`.

The above method builds the API documentation from the source code. Alternatively, you can do:

```
cd docs
make html
```

And the documentation will be generated in the same location, but using the *installed* version of Astropy.

3.3.6 Testing a source code build of Astropy

The easiest way to test that your Astropy built correctly (without installing astropy) is to run this from the root of the source tree:

```
python setup.py test
```

There are also alternative methods of *Running Tests*.

GETTING STARTED WITH ASTROPY

4.1 Importing Astropy

In order to encourage consistency amongst users in importing and using Astropy functionality, we have put together the following guidelines.

Since most of the functionality in Astropy resides in sub-packages, importing astropy as:

```
>>> import astropy
```

is not very useful. Instead, it is best to import the desired sub-package with the syntax:

```
>>> from astropy import subpackage
```

For example, to access the FITS-related functionality, you can import `astropy.io.fits` with:

```
>>> from astropy.io import fits
>>> hdulist = fits.open('data.fits')
```

In specific cases, we have recommended shortcuts in the documentation for specific sub-packages, for example:

```
>>> from astropy import units as u
>>> from astropy import coordinates as coord
>>> coord.SkyCoord(ra=10.68458*u.deg, dec=41.26917*u.deg, frame='icrs')
<SkyCoord (ICRS): ra=10.68458 deg, dec=41.26917 deg>
```

Finally, in some cases, most of the required functionality is contained in a single class (or a few classes). In those cases, the class can be directly imported:

```
>>> from astropy.cosmology import WMAP7
>>> from astropy.table import Table
>>> from astropy.wcs import WCS
```

Note that for clarity, and to avoid any issues, we recommend to **never** import any Astropy functionality using `*`, for example:

```
>>> from astropy.io.fits import * # NOT recommended
```

Some components of Astropy started off as standalone packages (e.g. PyFITS, PyWCS), so in cases where Astropy needs to be used as a drop-in replacement, the following syntax is also acceptable:

```
>>> from astropy.io import fits as pyfits
```

4.2 Getting started with subpackages

Because different subpackages have very different functionality, further suggestions for getting started are in the documentation for the subpackages, which you can reach by browsing the sections listed in the *User Documentation*.

Or, if you want to dive right in, you can either look at docstrings for particular a package or object, or access their documentation using the `find_api_page` function. For example, doing this:

```
>>> from astropy import find_api_page
>>> from astropy.units import Quantity
>>> find_api_page(Quantity)
```

Will bring up the documentation for the `Quantity` class in your browser.

Core data structures and transformations

CONSTANTS (ASTROPY.CONSTANTS)

5.1 Introduction

`astropy.constants` contains a number of physical constants useful in Astronomy. Constants are `Quantity` objects with additional meta-data describing their provenance and uncertainties.

5.2 Getting Started

To use the constants in S.I. units, you can import the constants directly from the `astropy.constants` sub-package:

```
>>> from astropy.constants import G
```

or, if you want to avoid having to explicitly import all the constants you need, you can simply do:

```
>>> from astropy import constants as const
```

and then subsequently use for example `const.G`. Constants are fully-fledged `Quantity` objects, so you can easily convert them to different units for example:

```
>>> print const.c
Name      = Speed of light in vacuum
Value     = 299792458.0
Error     = 0.0
Units     = m / s
Reference = CODATA 2010
```

```
>>> print const.c.to('km/s')
299792.458 km / s
```

```
>>> print const.c.to('pc/yr')
0.306601393788 pc / yr
```

and you can use them in conjunction with unit and other non-constant `Quantity` objects:

```
>>> from astropy import units as u
>>> F = (const.G * 3. * const.M_sun * 100 * u.kg) / (2.2 * u.au) ** 2
>>> print F.to(u.N)
0.367669392028 N
```

It is possible to convert most constants to cgs using e.g.:

```
>>> const.c.cgs
<Quantity 29979245800.0 cm / s>
```

However, some constants are defined with different physical dimensions in cgs and cannot be directly converted. Because of this ambiguity, such constants cannot be used in expressions without specifying a system:

```
>>> 100 * const.e
Traceback (most recent call last):
...
TypeError: Constant 'e' does not have physically compatible units
across all systems of units and cannot be combined with other
values without specifying a system (eg. e.emu)
>>> 100 * const.e.esu
<Quantity 4.8032045057134676e-08 Fr>
Traceback (most recent call last):
...
TypeError: Constant 'e' does not have physically compatible units
```

5.3 Reference/API

5.3.1 astropy.constants Module

Contains astronomical and physical constants for use in Astropy or other places.

A typical use case might be:

```
>>> from astropy.constants import c, m_e
>>> # ... define the mass of something you want the rest energy of as m ...
>>> m = m_e
>>> E = m * c**2
>>> E.to('MeV')
<Quantity 0.510998927603161 MeV>
```

The following constants are available:

Name	Value	Unit	Description
G	6.67384e-11	m ³ / (kg s ²)	Gravitational constant
L_sun	3.846e+26	W	Solar luminosity
M_earth	5.9742e+24	kg	Earth mass
M_jup	1.8987e+27	kg	Jupiter mass
M_sun	1.9891e+30	kg	Solar mass
N_A	6.02214129e+23	1 / (mol)	Avogadro's number
R	8.3144621	J / (K mol)	Gas constant
R_earth	6378136	m	Earth equatorial radius
R_jup	71492000	m	Jupiter equatorial radius
R_sun	695508000	m	Solar radius
Ryd	10973731.6	1 / (m)	Rydberg constant
a0	5.29177211e-11	m	Bohr radius
alpha	0.00729735257		Fine-structure constant
atmosphere	101325	Pa	Atmosphere
au	1.49597871e+11	m	Astronomical Unit
b_wien	0.0028977721	m K	Wien wavelength displacement law constant
c	299792458	m / (s)	Speed of light in vacuum
e	1.60217657e-19	C	Electron charge
eps0	8.85418782e-12	F/m	Electric constant
g0	9.80665	m / s ²	Standard acceleration of gravity
h	6.62606957e-34	J s	Planck constant

Continued on next page

Table 5.1 – continued from previous page

Name	Value	Unit	Description
hbar	1.05457173e-34	J s	Reduced Planck constant
k_B	1.3806488e-23	J / (K)	Boltzmann constant
kpc	3.08567758e+19	m	Kiloparsec
m_e	9.10938291e-31	kg	Electron mass
m_n	1.67492735e-27	kg	Neutron mass
m_p	1.67262178e-27	kg	Proton mass
mu0	1.25663706e-06	N/A ²	Magnetic constant
muB	9.27400968e-24	J/T	Bohr magneton
pc	3.08567758e+16	m	Parsec
sigma_sb	5.670373e-08	W / (K ⁴ m ²)	Stefan-Boltzmann constant
u	1.66053892e-27	kg	Atomic mass

Classes

<code>Constant(abbrev, name, value, unit, ...[, ...])</code>	A physical or astronomical constant.
<code>EMConstant(abbrev, name, value, unit, ...[, ...])</code>	An electromagnetic constant.

Constant

class `astropy.constants.Constant` (*abbrev, name, value, unit, uncertainty, reference, system=None*)

Bases: `astropy.units.quantity.Quantity`

A physical or astronomical constant.

These objects are quantities that are meant to represent physical constants.

Attributes Summary

<code>abbrev</code>	A typical ASCII text abbreviation of the constant, also generally the same as the Python variable used for this constant.
<code>cgs</code>	If the Constant is defined in the CGS system return that instance of the constant, else convert to a Quantity in the SI system.
<code>name</code>	The full name of the constant.
<code>reference</code>	The source used for the value of this constant.
<code>si</code>	If the Constant is defined in the SI system return that instance of the constant, else convert to a Quantity in the CGS system.
<code>system</code>	The system of units in which this constant is defined (typically <code>None</code> so long as the constant's units can be directly converted to the SI system).
<code>uncertainty</code>	The known uncertainty in this constant's value.
<code>unit</code>	The unit(s) in which this constant is defined.

Methods Summary

<code>copy()</code>	Return a copy of this <code>Constant</code> instance.
---------------------	---

Attributes Documentation

`abbrev`

A typical ASCII text abbreviation of the constant, also generally the same as the Python variable used for this constant.

cgs

If the Constant is defined in the CGS system return that instance of the constant, else convert to a Quantity in the appropriate CGS units.

name

The full name of the constant.

reference

The source used for the value of this constant.

si

If the Constant is defined in the SI system return that instance of the constant, else convert to a Quantity in the appropriate SI units.

system

The system of units in which this constant is defined (typically `None` so long as the constant's units can be directly converted between systems).

uncertainty

The known uncertainty in this constant's value.

unit

The unit(s) in which this constant is defined.

Methods Documentation

copy ()

Return a copy of this `Constant` instance. Since they are by definition immutable, this merely returns another reference to `self`.

EMConstant

class `astropy.constants.EMConstant` (*abbrev, name, value, unit, uncertainty, reference, system=None*)

Bases: `astropy.constants.Constant`

An electromagnetic constant.

Attributes Summary

`cgs` Overridden for `EMConstant` to raise a `TypeError` emphasizing that there are multiple EM extensions to CGS.

Attributes Documentation

cgs

Overridden for `EMConstant` to raise a `TypeError` emphasizing that there are multiple EM extensions to CGS.

Class Inheritance Diagram

UNITS AND QUANTITIES (ASTROPY.UNITS)

6.1 Introduction

`astropy.units` handles defining, converting between, and performing arithmetic with physical quantities. Examples of physical quantities are meters, seconds, Hz, etc.

`astropy.units` does not know spherical geometry or sexagesimal (hours, min, sec): if you want to deal with celestial coordinates, see the `astropy.coordinates` package.

6.2 Getting Started

Most users of the `astropy.units` package will work with “quantities”: the combination of a value and a unit. The easiest way to create a `Quantity` is to simply multiply or divide a value by one of the built-in units. It works with scalars, sequences and Numpy arrays:

```
>>> from astropy import units as u
>>> 42.0 * u.meter
<Quantity 42.0 m>
>>> [1., 2., 3.] * u.m
<Quantity [ 1., 2., 3.] m>
>>> import numpy as np
>>> np.array([1., 2., 3.]) * u.m
<Quantity [ 1., 2., 3.] m>
```

You can get the unit and value from a `Quantity` using the unit and value members:

```
>>> q = 42.0 * u.meter
>>> q.value
42.0
>>> q.unit
Unit("m")
```

From this simple building block, it's easy to start combining quantities with different units:

```
>>> 15.1 * u.meter / (32.0 * u.second)
<Quantity 0.47187... m / s>
>>> 3.0 * u.kilometer / (130.51 * u.meter / u.second)
<Quantity 0.0229867443... km s / m>
>>> (3.0 * u.kilometer / (130.51 * u.meter / u.second)).decompose()
<Quantity 22.9867443... s>
```

Unit conversion is done using the `to()` method, which returns a new `Quantity` in the given unit:

```
>>> x = 1.0 * u.parsec
>>> x.to(u.km)
<Quantity 30856775814671.9... km>
```

It is also possible to work directly with units at a lower level, for example, to create custom units:

```
>>> from astropy.units import imperial

>>> cms = u.cm / u.s
>>> # ...and then use some imperial units
>>> mph = imperial.mile / u.hour

>>> # And do some conversions
>>> q = 42.0 * cms
>>> q.to(mph)
<Quantity 0.93951324266284... mi / h>
```

Units that “cancel out” become a special unit called the “dimensionless unit”:

```
>>> u.m / u.m
Unit(dimensionless)
```

`astropy.units` is able to match compound units against the units it already knows about:

```
>>> (u.s ** -1).compose()
[Unit("Bq"), Unit("Hz"), Unit("3.7e+10 Ci")]
```

And it can convert between unit systems, such as SI or CGS:

```
>>> (1.0 * u.Pa).cgs
<Quantity 10.0 Ba>
```

`astropy.units` also handles equivalencies, such as that between wavelength and frequency. To use that feature, equivalence objects are passed to the `to()` conversion method. For instance, a conversion from wavelength to frequency doesn’t normally work:

```
>>> (1000 * u.nm).to(u.Hz)
Traceback (most recent call last):
...
UnitsError: 'nm' (length) and 'Hz' (frequency) are not convertible
Traceback (most recent call last):
...
UnitsError: 'nm' (length) and 'Hz' (frequency) are not convertible
```

but by passing an equivalency list, in this case `spectral()`, it does:

```
>>> (1000 * u.nm).to(u.Hz, equivalencies=u.spectral())
<Quantity 299792457999999.94 Hz>
```

Quantities and units can be printed nicely to strings using the [Format String Syntax](#), the preferred string formatting syntax in recent versions of python. Format specifiers (like `0.03f`) in new-style format strings will be used to format the quantity value:

```
>>> q = 15.1 * u.meter / (32.0 * u.second)
>>> q
<Quantity 0.47187... m / s>
>>> "{0:0.03f}".format(q)
'0.472 m / s'
```

The value and unit can also be formatted separately. Format specifiers used on units can be used to choose the unit formatter:

```
>>> q = 15.1 * u.meter / (32.0 * u.second)
>>> q
<Quantity 0.47187... m / s>
>>> "{0.value:0.03f} {0.unit:FITS}".format(q)
'0.472 m s-1'
```

6.3 Using `astropy.units`

6.3.1 Quantity

The `Quantity` object is meant to represent a value that has some unit associated with the number.

Creating `Quantity` instances

`Quantity` objects are created through multiplication or division with `Unit` objects. For example, to create a `Quantity` to represent 15 m/s:

```
>>> import astropy.units as u
>>> 15 * u.m / u.s
<Quantity 15.0 m / s>
```

Note: `Quantity` objects are converted to float by default.

As another example:

```
>>> 1.25 / u.s
<Quantity 1.25 1 / s>
```

You can also create instances using the `Quantity` constructor directly, by specifying a value and unit:

```
>>> u.Quantity(15, u.m / u.s)
<Quantity 15.0 m / s>
```

`Quantity` objects can also be created automatically from Numpy arrays or Python sequences:

```
>>> [1, 2, 3] * u.m
<Quantity [ 1., 2., 3.] m>
>>> import numpy as np
>>> np.array([1, 2, 3]) * u.m
<Quantity [ 1., 2., 3.] m>
```

`Quantity` objects can also be created from sequences of `Quantity` objects, as long as all of their units are equivalent, and will automatically convert to Numpy arrays:

```
>>> qlst = [60 * u.s, 1 * u.min]
>>> u.Quantity(qlst, u.minute)
<Quantity [ 1., 1.] min>
```

Finally, the current unit and value can be accessed via the `unit` and `value` attributes:

```
>>> q = 2.5 * u.m / u.s
>>> q.unit
Unit("m / s")
>>> q.value
2.5
```

Converting to different units

`Quantity` objects can be converted to different units using the `to()` method:

```
>>> q = 2.3 * u.m / u.s
>>> q.to(u.km / u.h)
<Quantity 8.2... km / h>
```

For convenience, the `si` and `cgs` attributes can be used to convert the `Quantity` to base S.I. or c.g.s units:

```
>>> q = 2.4 * u.m / u.s
>>> q.si
<Quantity 2... m / s>
>>> q.cgs
<Quantity 240.0 cm / s>
```

Arithmetic

Addition and Subtraction

Addition or subtraction between `Quantity` objects is supported when their units are equivalent. When the units are equal, the resulting object has the same unit:

```
>>> 11 * u.s + 30 * u.s
<Quantity 41.0 s>
>>> 30 * u.s - 11 * u.s
<Quantity 19.0 s>
```

If the units are equivalent, but not equal (e.g. kilometer and meter), the resulting object **has units of the object on the left**:

```
>>> 1100.1 * u.m + 13.5 * u.km
<Quantity 14600.1 m>
>>> 13.5 * u.km + 1100.1 * u.m
<Quantity 14.600... km>
>>> 1100.1 * u.m - 13.5 * u.km
<Quantity -12399.9 m>
>>> 13.5 * u.km - 1100.1 * u.m
<Quantity 12.399... km>
```

Addition and subtraction is not supported between `Quantity` objects and basic numeric types:

```
>>> 13.5 * u.km + 19.412
Traceback (most recent call last):
...
UnitsError: Can only apply 'add' function to dimensionless
quantities when other argument is not a quantity (unless the
latter is all zero/infinity/nan)
Traceback (most recent call last):
...
UnitsError: Can only apply 'add' function to dimensionless
```

except for dimensionless quantities (see [Dimensionless quantities](#)).

Multiplication and Division

Multiplication and division is supported between `Quantity` objects with any units, and with numeric types. For these operations between objects with equivalent units, the **resulting object has composite units**:

```
>>> 1.1 * u.m * 140.3 * u.cm
<Quantity 154.33... cm m>
>>> 140.3 * u.cm * 1.1 * u.m
<Quantity 154.33... cm m>
>>> 1. * u.m / (20. * u.cm)
<Quantity 0.05... m / cm>
>>> 20. * u.cm / (1. * u.m)
<Quantity 20.0 cm / m>
```

For multiplication, you can change how to represent the resulting object by using the `to()` method:

```
>>> (1.1 * u.m * 140.3 * u.cm).to(u.m**2)
<Quantity 1.5433... m2>
>>> (1.1 * u.m * 140.3 * u.cm).to(u.cm**2)
<Quantity 15433.0... cm2>
```

For division, if the units are equivalent, you may want to make the resulting object dimensionless by reducing the units. To do this, use the `decompose()` method:

```
>>> (20. * u.cm / (1. * u.m)).decompose()
<Quantity 0.2...>
```

This method is also useful for more complicated arithmetic:

```
>>> 15. * u.kg * 32. * u.cm * 15 * u.m / (11. * u.s * 1914.15 * u.ms)
<Quantity 0.341950972... cm kg m / (ms s)>
>>> (15. * u.kg * 32. * u.cm * 15 * u.m / (11. * u.s * 1914.15 * u.ms)).decompose()
<Quantity 3.41950972... kg m2 / s2>
```

Numpy functions

`Quantity` objects are actually full Numpy arrays (the `Quantity` object class inherits from and extends the `numpy.ndarray` class), and we have tried to ensure that most Numpy functions behave properly with quantities:

```
>>> q = np.array([1., 2., 3., 4.]) * u.m / u.s
>>> np.mean(q)
<Quantity 2.5 m / s>
>>> np.std(q)
<Quantity 1.118033... m / s>
```

including functions that only accept specific units such as angles:

```
>>> q = 30. * u.deg
>>> np.sin(q)
<Quantity 0.4999999...>
```

or dimensionless quantities:

```
>>> from astropy.constants import h, k_B
>>> nu = 3 * u.GHz
>>> T = 30 * u.K
>>> np.exp(-h * nu / (k_B * T))
<Quantity 0.99521225...>
```

(see [Dimensionless quantities](#) for more details).

Dimensionless quantities

Dimensionless quantities have the characteristic that if they are added or subtracted from a Python scalar or unitless `ndarray`, or if they are passed to a Numpy function that takes dimensionless quantities, the units are simplified so that the quantity is dimensionless and scale-free. For example:

```
>>> 1. + 1. * u.m / u.km
<Quantity 1.00...>
```

which is different from:

```
>>> 1. + (1. * u.m / u.km).value
2.0
```

In the latter case, the result is 2.0 because the unit of `(1. * u.m / u.km)` is not scale-free by default:

```
>>> q = (1. * u.m / u.km)
>>> q.unit
Unit("m / km")
>>> q.unit.decompose()
Unit(dimensionless with a scale of 0.001)
```

However, when combining with a non-quantity object, the unit is automatically decomposed to be scale-free, giving the expected result.

This also occurs when passing dimensionless quantities to functions that take dimensionless quantities:

```
>>> nu = 3 * u.GHz
>>> T = 30 * u.K
>>> np.exp(- h * nu / (k_B * T))
<Quantity 0.99521225...>
```

The result is independent from the units the different quantities were specified in:

```
>>> nu = 3.e9 * u.Hz
>>> T = 30 * u.K
>>> np.exp(- h * nu / (k_B * T))
<Quantity 0.99521225...>
```

Converting to plain Python scalars

Converting `Quantity` objects does not work for non-dimensionless quantities:

```
>>> float(3. * u.m)
Traceback (most recent call last):
...
TypeError: Only dimensionless scalar quantities can be converted
to Python scalars
Traceback (most recent call last):
...
TypeError: Only dimensionless scalar quantities can be converted
```

Instead, only dimensionless values can be converted to plain Python scalars:

```
>>> float(3. * u.m / (4. * u.m))
0.75
>>> float(3. * u.km / (4. * u.m))
750.0
>>> int(6. * u.km / (2. * u.m))
3000
```

Known issues with conversion to numpy arrays

Since `Quantity` objects are Numpy arrays, we are not able to ensure that only dimensionless quantities are converted to Numpy arrays:

```
>>> np.array([1, 2, 3] * u.m)
array([ 1.,  2.,  3.]
```

Similarly, while most numpy functions work properly, a few have *known issues*, either ignoring the unit (e.g., `np.dot`) or not reinitializing it properly (e.g., `np.hstack`). This propagates to more complex functions such as `np.linalg.norm` and `scipy.integrate.odeint`.

Subclassing Quantity

To subclass `Quantity`, one generally proceeds as one would when subclassing `ndarray`, i.e., one typically needs to override `__new__` (rather than `__init__`) and uses the `numpy.ndarray.__array_finalize__` method to update attributes. For details, see the [numpy documentation on subclassing](#). For examples, one can look at `Quantity` itself, where, e.g., the `astropy.units.Quantity.__array_finalize__` method is used to pass on the unit, at `Angle`, where strings are parsed as angles in the `astropy.coordinates.Angle.__new__` method and at `Longitude`, where the `astropy.coordinates.Longitude.__array_finalize__` method is used to pass on the angle at which longitudes wrap.

Another method that is meant to be overridden by subclasses, one specific to `Quantity`, is `astropy.units.Quantity.__quantity_subclass__`. This is called to decide which type of subclass to return, based on the unit of the quantity that is to be created. It is used, e.g., in `Angle` to return a `Quantity` if a calculation returns a unit other than an angular one.

6.3.2 Standard units

Standard units are defined in the `astropy.units` package as object instances.

All units are defined in term of basic ‘irreducible’ units. The irreducible units include:

- Length (meter)
- Time (second)
- Mass (kilogram)
- Current (ampere)
- Temperature (Kelvin)
- Angular distance (radian)
- Solid angle (steradian)
- Luminous intensity (candela)
- Stellar magnitude (mag)

- Amount of substance (mole)
- Photon count (photon)

(There are also some more obscure base units required by the FITS standard that are no longer recommended for use.)

Units that involve combinations of fundamental units are instances of `CompositeUnit`. In most cases, one does not need to worry about the various kinds of unit classes unless one wants to design a more complex case.

There are many units already predefined in the module. One may use the `find_equivalent_units` method to list all the existing predefined units of a given type:

```
>>> from astropy import units as u
>>> u.g.find_equivalent_units()
Primary name | Unit definition | Aliases
[
  M_e        | 9.10938e-31 kg |
  M_p        | 1.67262e-27 kg |
  g          | 0.001 kg      | gram
  kg         | irreducible   | kilogram
  solMass    | 1.9891e+30 kg | M_sun, Msun
  t          | 1000 kg       | tonne
  u          | 1.66054e-27 kg | Da, Dalton
]
```

The dimensionless unit

In addition to these units, `astropy.units` includes the concept of the dimensionless unit, used to indicate quantities that don't have a physical dimension. This is distinct in concept from a unit that is equal to `None`: that indicates that no unit was specified in the data or by the user.

For convenience, there is a unit that is both dimensionless and unscaled: the `dimensionless_unscaled` object:

```
>>> from astropy import units as u
>>> u.dimensionless_unscaled
Unit(dimensionless)
```

Dimensionless quantities are often defined as products or ratios of quantities that are not dimensionless, but whose dimensions cancel out when their powers are multiplied. For example:

```
>>> u.m / u.m
Unit(dimensionless)
```

For compatibility with the supported unit string formats, this is equivalent to `Unit('')` and `Unit(1)`, though using `u.dimensionless_unscaled` in Python code is preferred for readability:

```
>>> u.dimensionless_unscaled == u.Unit('')
True
>>> u.dimensionless_unscaled == u.Unit(1)
True
```

Note that in many cases, a dimensionless unit may also have a scale. For example:

```
>>> (u.km / u.m).decompose()
Unit(dimensionless with a scale of 1000.0)
>>> (u.km / u.m).decompose() == u.dimensionless_unscaled
False
```

To determine if a unit is dimensionless (but regardless of the scale), use the `physical_type` property:

```

>>> (u.km / u.m).physical_type
u'dimensionless'
>>> # This also has a scale, so it is not the same as u.dimensionless_unscaled
>>> (u.km / u.m) == u.dimensionless_unscaled
False
>>> # However, (u.m / u.m) has a scale of 1.0, so it is the same
>>> (u.m / u.m) == u.dimensionless_unscaled
True

```

Enabling other units

By default, only the “default” units are searched by `find_equivalent_units` and similar methods that do searching. This includes SI, CGS and astrophysical units. However, one may wish to enable the imperial or other user-defined units.

For example, to enable Imperial units, simply do:

```

>>> from astropy.units import imperial
>>> imperial.enable()
>>> u.m.find_equivalent_units()
  Primary name | Unit definition | Aliases
[
  AU           | 1.49598e+11 m  | au           ,
  Angstrom    | 1e-10 m        | AA, angstrom ,
  cm          | 0.01 m         | centimeter  ,
  ft          | 0.3048 m       | foot        ,
  inch        | 0.0254 m       |             ,
  lyr         | 9.46073e+15 m  | lightyear   ,
  m           | irreducible    | meter       ,
  mi          | 1609.34 m      | mile        ,
  micron      | 1e-06 m        |             ,
  nmi         | 1852 m         | nauticalmile, NM ,
  pc          | 3.08568e+16 m  | parsec      ,
  solRad      | 6.95508e+08 m  | R_sun       ,
  yd          | 0.9144 m       | yard        ,
]

```

This may also be used with the `with` statement, to temporarily enable additional units:

```

>>> from astropy import units as u
>>> from astropy.units import imperial
>>> with imperial.enable():
...     u.m.find_equivalent_units()
...

```

To enable just specific units, use `add_enabled_units`:

```

>>> from astropy import units as u
>>> from astropy.units import imperial
>>> with u.add_enabled_units_context([imperial.knot]):
...     u.m.find_equivalent_units()
...

```

6.3.3 Combining and defining units

Units and quantities can be combined together using the regular Python numeric operators. For example:

```
>>> from astropy import units as u
>>> fluxunit = u.erg / (u.cm ** 2 * u.s)
>>> fluxunit
Unit("erg / (cm2 s)")
>>> 52.0 * fluxunit
<Quantity 52.0 erg / (cm2 s)>
>>> 52.0 * fluxunit / u.s
<Quantity 52.0 erg / (cm2 s2)>
```

Units support fractional powers, which retain their precision through complex operations. To do this, it is recommended to use `fractions.Fraction` objects. For example:

```
>>> from astropy.utils.compat.fractions import Fraction
>>> Franklin = u.g ** Fraction(1, 2) * u.cm ** Fraction(3, 2) * u.s ** -1
```

Note: Floating-point powers that are effectively the same as fractions with a denominator less than 10 are implicitly converted to `Fraction` objects under the hood. Therefore the following are equivalent:

```
>>> x = u.m ** Fraction(1, 3)
>>> x.powers
[Fraction(1, 3)]
>>> x = u.m ** (1. / 3.)
>>> x.powers
[Fraction(1, 3)]
```

Users are free to define new units, either fundamental or compound using the `def_unit` function. For example:

```
>>> bakers_fortnight = u.def_unit('bakers_fortnight', 13 * u.day)
```

The addition of a string gives the new unit a name that will show up when the unit is printed.

Creating a new fundamental unit is simple:

```
>>> titter = u.def_unit('titter')
>>> chuckle = u.def_unit('chuckle', 5 * titter)
>>> laugh = u.def_unit('laugh', 4 * chuckle)
>>> guffaw = u.def_unit('guffaw', 3 * laugh)
>>> rofl = u.def_unit('rofl', 4 * guffaw)
>>> death_by_laughing = u.def_unit('death_by_laughing', 10 * rofl)
>>> rofl.to(titter, 1)
240.0
```

By default, custom units are not searched by methods such as `find_equivalent_units`. However, they can be enabled by calling `add_enabled_units`:

```
>>> kmph = u.def_unit('kmph', u.km / u.h)
>>> (u.m / u.s).find_equivalent_units()
[]
>>> u.add_enabled_units([kmph])
<astropy.units.core._UnitContext object at ...>
>>> (u.m / u.s).find_equivalent_units()
  Primary name | Unit definition | Aliases
[
  kmph         | 0.277778 m / s |
]
```

6.3.4 Decomposing and composing units

Reducing a unit to its irreducible parts

A unit or quantity can be decomposed into its irreducible parts using the `Unit.decompose` or `Quantity.decompose` methods:

```
>>> from astropy import units as u
>>> u.Ry
Unit("Ry")
>>> u.Ry.decompose()
Unit("2.17987e-18 kg m2 / s2")
```

You can limit the selection of units that you want to decompose to using the `bases` keyword argument:

```
>>> u.Ry.decompose(bases=[u.m, u.N])
Unit("2.17987e-18 m N")
```

This is also useful to decompose to a particular system. For example, to decompose the Rydberg unit in terms of CGS units:

```
>>> u.Ry.decompose(bases=u.cgs.bases)
Unit("2.17987e-11 cm2 g / s2")
```

Automatically composing a unit into more complex units

Conversely, a unit may be recomposed back into more complex units using the `compose` method. Since there may be multiple equally good results, a list is always returned:

```
>>> x = u.Ry.decompose()
>>> x.compose()
[Unit("Ry"),
 Unit("2.17987e-18 J"),
 Unit("2.17987e-11 erg"),
 Unit("13.6057 eV")]
```

Some other interesting examples:

```
>>> (u.s ** -1).compose()
[Unit("Bq"), Unit("Hz"), Unit("3.7e+10 Ci")]
```

Composition can be combined with *Equivalencies*:

```
>>> (u.s ** -1).compose(equivalencies=u.spectral())
[Unit("m"),
 Unit("Hz"),
 Unit("J"),
 Unit("Bq"),
 Unit("3.24078e-17 pc"),
 Unit("1.057e-16 lyr"),
 Unit("6.68459e-12 AU"),
 Unit("1.4378e-09 solRad"),
 Unit("0.01 k"),
 Unit("100 cm"),
 Unit("1e+06 micron"),
 Unit("1e+07 erg"),
 Unit("1e+10 Angstrom"),
 Unit("3.7e+10 Ci"),
```

```
Unit("4.58743e+17 Ry"),  
Unit("6.24151e+18 eV")]
```

Obviously a name doesn't exist for every arbitrary derived unit imaginable. In that case, the system will do its best to reduce the unit to the fewest possible symbols:

```
>>> (u.cd * u.sr * u.V * u.s).compose()  
[Unit("lm Wb")]
```

Converting between systems

Built on top of this functionality is a convenience method to convert between unit systems.

```
>>> u.Pa.to_system(u.cgs)  
[Unit("10 Ba")]
```

There is also a shorthand for this which only returns the first of many possible matches:

```
>>> u.Pa.cgs  
Unit("10 Ba")
```

This is equivalent to decomposing into the new system and then composing into the most complex units possible, though `to_system` adds some extra logic to return the results sorted in the most useful order:

```
>>> u.Pa.decompose(bases=u.cgs.bases)  
Unit("10 g / (cm s2)")  
>>> _.compose(units=u.cgs)  
[Unit("10 Ba")]
```

6.3.5 String representations of units

Converting units to string representations

You can control the way that `Quantity` and `UnitBase` objects are rendered as strings using the new [Format String Syntax](#). New-style format strings use the `"{}".format()` syntax. Most of the format specifiers are similar to the old `%`-style formatting, so things like `0.003f` still work, just in the form `"{:0.003f}".format()`.

For quantities, format specifiers, like `0.003f` will be applied to the `Quantity` value, without affecting the unit. Specifiers like `20s`, which would only apply to a string, will be applied to the whole string representation of the `Quantity`. This means you can do:

```
>>> from astropy import units as u  
>>> import numpy as np  
>>> q = 10. * u.km  
>>> q  
<Quantity 10.0 km>  
>>> "{}".format(q)  
'10.0 km'  
>>> "{0:+0.03f}".format(q)  
'+10.000 km'  
>>> "{0:20s}".format(q)  
'10.0 km'
```

To format both the value and the unit separately, you can access the `Quantity` class attributes within new-style format strings:

```
>>> q = 10. * u.km
>>> q
<Quantity 10.0 km>
>>> "{0.value:0.003f} in {0.unit:s}".format(q)
'10.000 in km'
```

Because Numpy arrays don't accept most format specifiers, using specifiers like `0.003f` will not work when applied to a Numpy array or non-scalar `Quantity`. Use `numpy.array_str()` instead. For example:

```
>>> q = np.linspace(0,1,10) * u.m
>>> "{0} {1}".format(np.array_str(q.value, precision=1), q.unit)
'[ 0.  0.1  0.2  0.3  0.4  0.6  0.7  0.8  0.9  1. ] m'
```

Examine the numpy documentation for more examples with `numpy.array_str()`.

Units, or the unit part of a quantity, can also be formatted in a number of different styles. By default, the string format used is referred to as the “generic” format, which is based on syntax of the FITS standard's format for representing units, but supports all of the units defined within the `astropy.units` framework, including user-defined units. The format specifier (and `to_string`) functions also take an optional parameter to select a different format, including “`latex`”, “`unicode`”, “`cds`”, and others, defined below.

```
>>> "{0.value:0.003f} in {0.unit:latex}".format(q)
'10.000 in  $\mathrm{km}$ '
>>> fluxunit = u.erg / (u.cm ** 2 * u.s)
>>> "{0}".format(fluxunit)
u'erg / (cm2 s)'
>>> print("{0:console}".format(fluxunit))
  erg
-----
s cm^2
>>> "{0:latex}".format(fluxunit)
u' $\mathrm{\frac{erg}{s\,cm^2}}$ '
>>> "{0:>20s}".format(fluxunit)
u'          erg / (cm2 s)'
```

The `to_string` method is an alternative way to format units as strings, and is the underlying implementation of the `format-style` usage:

```
>>> fluxunit = u.erg / (u.cm ** 2 * u.s)
>>> fluxunit.to_string('latex')
u' $\mathrm{\frac{erg}{s\,cm^2}}$ '
```

Creating units from strings

Units can also be created from strings in a number of different formats using the `Unit` class:

```
>>> from astropy import units as u
>>> u.Unit("m")
Unit("m")
>>> u.Unit("erg / (s cm2)")
Unit("erg / (cm2 s)")
>>> u.Unit("erg.s-1.cm-2", format="cds")
Unit("erg / (cm2 s)')
```

Note: Creating units from strings requires the use of a specialized parser for the unit language, which results in a performance penalty if units are created using strings. Thus, it is much faster to use unit objects directly (e.g., `unit = u.degree / u.minute`) instead of via string parsing (`unit = u.Unit('deg/min')`). This parser is very useful, however, if your unit definitions are coming from a file format such as FITS or VOTable.

Built-in formats

`astropy.units` includes support for parsing and writing the following formats:

- "fits": This is the format defined in the Units section of the [FITS Standard](#). Unlike the “generic” string format, this will only accept or generate units defined in the FITS standard.
- "vounit": The [proposed IVOA standard](#) for representing units in the VO. Again, based on the FITS syntax, but the collection of supported units is different.
- "cds": [Standards for astronomical catalogues from Centre de Données astronomiques de Strasbourg](#): This is the standard used by [Vizier tables](#), as well as what is used by VOTable versions 1.2 and earlier.
- "ogip": A standard for storing units as recommended by the [Office of Guest Investigator Programs \(OGIP\)](#).

`astropy.units` is also able to write, but not read, units in the following formats:

- "latex": Writes units out using LaTeX math syntax using the [IAU Style Manual](#) recommendations for unit presentation. This format is automatically used when printing a unit in the IPython notebook:

```
>>> fluxunit
```

$$\frac{\text{erg}}{\text{s cm}^2}$$

- "console": Writes a multi-line representation of the unit useful for display in a text console:

```
>>> print fluxunit.to_string('console')
erg
-----
s cm^2
```

- "unicode": Same as "console", except uses Unicode characters:

```
>>> print u.Ry.decompose().to_string('unicode')
      m2 kg
2.1798721×10-18 -----
                    s2
```

Unrecognized Units

Since many files in found in the wild have unit strings that do not correspond to any given standard, `astropy.units` also has a consistent way to store and pass around unit strings that did not parse.

Normally, passing an unrecognized unit string raises an exception:

```
>>> # The FITS standard uses 'angstrom', not 'Angstroem'
>>> u.Unit("Angstroem", format="fits")
Traceback (most recent call last):
...
ValueError: 'Angstroem' did not parse as fits unit: At col 0, Unit
u'Angstroem' not supported by the FITS standard. Did you mean
Angstrom or angstrom?
Traceback (most recent call last):
...
ValueError: 'Angstroem' did not parse as fits unit: At col 0, Unit
```

However, the `Unit` constructor has the keyword argument `parse_strict` that can take one of three values to control this behavior:

- `'raise'`: (default) raise a `ValueError` exception.
- `'warn'`: emit a `Warning`, and return an `UnrecognizedUnit` instance.
- `'silent'`: return an `UnrecognizedUnit` instance.

So, for example, one can do:

```
>>> x = u.Unit("Angstroem", format="fits", parse_strict="warn")
WARNING: UnitsWarning: 'Angstroem' did not parse as unit format
'fits': At col 0, 'Angstroem' is not a valid unit in string
'Angstroem' [astropy.units.core]
```

This `UnrecognizedUnit` object remembers the original string it was created with, so it can be written back out, but any meaningful operations on it, such as converting to another unit or composing with other units, will fail.

```
>>> x.to_string()
'Angstroem'
>>> x.to(u.km)
Traceback (most recent call last):
...
ValueError: The unit 'Angstroem' is unrecognized. It can not be
converted to other units.
>>> x / u.m
Traceback (most recent call last):
...
ValueError: The unit 'Angstroem' is unrecognized, so all arithmetic
operations with it are invalid.
Traceback (most recent call last):
...
ValueError: The unit 'Angstroem' is unrecognized, so all arithmetic
```

6.3.6 Equivalencies

The unit module has machinery for supporting equivalencies between different units in certain contexts. Namely when equations can uniquely relate a value in one unit to a different unit. A good example is the equivalence between wavelength, frequency and energy for specifying a wavelength of radiation. Normally these units are not convertible, but when understood as representing light, they are convertible in certain contexts. This will describe how to use the equivalencies included in `astropy.units` and then describe how to define new equivalencies.

Equivalencies are used by passing a list of equivalency pairs to the `equivalencies` keyword argument of `Quantity.to`, `Unit.to` or `Unit.get_converter` methods. Alternatively, if a larger piece of code needs the same equivalencies, one can set them for a *given context*.

Built-in equivalencies

Parallax Units

`parallax()` is a function that returns an equivalency list to handle conversions between angles and length.

Length and angles are not normally convertible, so `to()` raises an exception:

```
>>> from astropy import units as u
>>> (8.0 * u.arcsec).to(u.parsec)
```

```
Traceback (most recent call last):
...
UnitsError: 'arcsec' (angle) and 'pc' (length) are not convertible
Traceback (most recent call last):
...
UnitsError: 'arcsec' (angle) and 'pc' (length) are not convertible
```

However, when passing the result of `parallax()` as the third argument to the `to()` method, angles can be converted into units of length (and vice versa).

```
>>> (8.0 * u.arcsec).to(u.parsec, equivalencies=u.parallax())
<Quantity 0.125 pc>
>>> u.AU.to(u.arcminute, equivalencies=u.parallax())
3437.7467707580054
```

Angles as Dimensionless Units

Angles are treated as a physically distinct type, which usually helps to avoid mistakes. For units such as rotational energy, however, it is not very handy. (Indeed, this double-sidedness underlies why radian went from [supplementary to derived unit](#).) The function `dimensionless_angles()` provides the required equivalency list that helps convert between angles and dimensionless units. It is somewhat different from all others in that it allows an arbitrary change in the number of powers to which radian is raised (i.e., including zero and thus dimensionless). For instance, normally the following raise exceptions:

```
>>> from astropy import units as u
>>> u.degree.to('')
Traceback (most recent call last):
...
UnitsError: 'deg' (angle) and '' (dimensionless) are not convertible
>>> (u.kg * u.m**2 * (u.cycle / u.s)**2).to(u.J)
Traceback (most recent call last):
...
UnitsError: 'cycle2 kg m2 / s2' and 'J' (energy) are not convertible
Traceback (most recent call last):
...
UnitsError: 'cycle2 kg m2 / s2' and 'J' (energy) are not convertible
```

But when passing we pass the proper conversion function, `dimensionless_angles()`, it works.

```
>>> u.deg.to('', equivalencies=u.dimensionless_angles())
0.01745329...
>>> (0.5e38 * u.kg * u.m**2 * (u.cycle / u.s)**2).to(u.J,
...          equivalencies=u.dimensionless_angles())
<Quantity 1.97392...e+39 J>
>>> import numpy as np
>>> np.exp(1j*0.125*u.cycle).to('', equivalencies=u.dimensionless_angles())
<Quantity (0.707106781186...+0.707106781186...j)>
```

The example with complex numbers is also one may well be doing a fair number of similar calculations. For such situations, there is the option to *set default equivalencies*.

Spectral Units

`spectral()` is a function that returns an equivalency list to handle conversions between wavelength, frequency, energy, and wave number.

As mentioned above with parallax units, we simply pass a list of equivalencies (in this case, the result of `spectral()`) as the third argument to the `to()` method and wavelength, frequency and energy can be converted.

```
>>> ([1000, 2000] * u.nm).to(u.Hz, equivalencies=u.spectral())
<Quantity [ 2.99792458e+14, 1.49896229e+14] Hz>
>>> ([1000, 2000] * u.nm).to(u.eV, equivalencies=u.spectral())
<Quantity [ 1.239..., 0.619...] eV>
```

These equivalencies even work with non-base units:

```
>>> # Inches to calories
>>> from astropy.units import imperial
>>> imperial.inch.to(imperial.Cal, equivalencies=u.spectral())
1.8691807591...e-27
```

Spectral (Doppler) equivalencies

Spectral equivalencies allow you to convert between wavelength, frequency, energy, and wave number but not to velocity, which is frequently the quantity of interest.

It is fairly straightforward to define the equivalency, but note that there are different *conventions*. In these conventions f_0 is the rest frequency, f is the observed frequency, V is the velocity, and c is the speed of light:

- Radio $V = c \frac{f_0 - f}{f_0}$; $f(V) = f_0(1 - V/c)$
- Optical $V = c \frac{f_0 - f}{f}$; $f(V) = f_0(1 + V/c)^{-1}$
- Relativistic $V = c \frac{f_0^2 - f^2}{f_0^2 + f^2}$; $f(V) = f_0 \frac{(1 - (V/c)^2)^{1/2}}{(1 + V/c)}$

These three conventions are implemented in `astropy.units.equivalencies` as `doppler_optical()`, `doppler_radio()`, and `doppler_relativistic()`. Example use:

```
>>> restfreq = 115.27120 * u.GHz # rest frequency of 12 CO 1-0 in GHz
>>> freq_to_vel = u.doppler_radio(restfreq)
>>> (116e9 * u.Hz).to(u.km / u.s, equivalencies=freq_to_vel)
<Quantity -1895.4321928669085 km / s>
```

Spectral Flux Density Units

There is also support for spectral flux density units. Their use is more complex, since it is necessary to also supply the location in the spectrum for which the conversions will be done, and the units of those spectral locations. The function that handles these unit conversions is `spectral_density()`. This function takes as its arguments the `Quantity` for the spectral location. For example:

```
>>> (1.5 * u.Jy).to(u.erg / u.cm**2 / u.s / u.Hz,
...               equivalencies=u.spectral_density(3500 * u.AA))
<Quantity 1.5e-23 erg / (cm2 Hz s)>
>>> (1.5 * u.Jy).to(u.erg / u.cm**2 / u.s / u.micron,
...               equivalencies=u.spectral_density(3500 * u.AA))
<Quantity 3.670928057142...e-08 erg / (cm2 micron s)>
```

Brightness Temperature / Flux Density Equivalency

There is an equivalency for brightness temperature and flux density. This equivalency is often referred to as “Antenna Gain” since, at a given frequency, telescope brightness sensitivity is unrelated to aperture size, but flux density

sensitivity is, so this equivalency is only dependent on the aperture size. See [Tools of Radio Astronomy](#) for details.

The `brightness_temperature` equivalency requires the beam area and frequency as arguments. Example:

```
>>> import numpy as np
>>> omega_B = np.pi * (50 * u.arcsec)**2
>>> freq = 5 * u.GHz
>>> u.Jy.to(u.K, equivalencies=u.brightness_temperature(omega_B, freq))
7.052588858...
```

Temperature Energy Equivalency

This equivalency allows conversion between temperature and its equivalent in energy (i.e., the temperature multiplied by the Boltzmann constant), usually expressed in electronvolts. This is used frequently for observations at high-energy, be it for solar or X-ray astronomy. Example:

```
>>> import astropy.units as u
>>> t_k = 1e6 * u.K
>>> t_k.to(u.eV, equivalencies=u.temperature_energy())
<Quantity 86.17332384... eV>
```

Writing new equivalencies

An equivalence list is just a list of tuples, where each tuple has 4 elements:

```
(from_unit, to_unit, forward, backward)
```

`from_unit` and `to_unit` are the equivalent units. `forward` and `backward` are functions that convert values between those units.

For example, until 1964 the metric liter was defined as the volume of 1kg of water at 4°C at 760mm mercury pressure. Volumes and masses are not normally directly convertible, but if we hold the constants in the 1964 definition of the liter as true, we could build an equivalency for them:

```
>>> liters_water = [
...     (u.l, u.g, lambda x: 1000.0 * x, lambda x: x / 1000.0)
... ]
>>> u.l.to(u.kg, 1, equivalencies=liters_water)
1.0
```

Note that the equivalency can be used with any other compatible units:

```
>>> from astropy.units import imperial
>>> imperial.gallon.to(imperial.pound, 1, equivalencies=liters_water)
8.3454044633335...
```

And it also works in the other direction:

```
>>> imperial.lb.to(imperial.pint, 1, equivalencies=liters_water)
0.9586114172355...
```

A slightly more complicated example: Spectral Doppler Equivalencies

We show how to define an equivalency using the radio convention for CO 1-0. This function is already defined in `doppler_radio()`, but this example is illustrative:

```
>>> from astropy.constants import si
>>> restfreq = 115.27120 # rest frequency of 12 CO 1-0 in GHz
>>> freq_to_vel = [(u.GHz, u.km/u.s,
... lambda x: (restfreq-x) / restfreq * si.c.to('km/s').value,
... lambda x: (1-x/si.c.to('km/s').value) * restfreq)]
>>> u.Hz.to(u.km / u.s, 116e9, equivalencies=freq_to_vel)
-1895.432192...
>>> (116e9 * u.Hz).to(u.km / u.s, equivalencies=freq_to_vel)
<Quantity -1895.432192... km / s>
```

Note that once this is defined for GHz and km/s, it will work for all other units of frequency and velocity. `x` is converted from the input frequency unit (e.g., Hz) to GHz before being passed to `lambda x:`. Similarly, the return value is assumed to be in units of km/s, which is why the `.value` of `c` is used instead of the constant.

Displaying available equivalencies

The `find_equivalent_units()` method also understands equivalencies. For example, without passing equivalencies, there are three compatible units for Hz in the standard set:

```
>>> u.Hz.find_equivalent_units()
Primary name | Unit definition | Aliases
[
  Bq          | 1 / s          | becquerel ,
  Ci          | 2.7027e-11 / s | curie     ,
  Hz          | 1 / s          | Hertz, hertz ,
]
```

However, when passing the spectral equivalency, you can see there are all kinds of things that Hz can be converted to:

```
>>> u.Hz.find_equivalent_units(equivalencies=u.spectral())
Primary name | Unit definition | Aliases
[
  AU          | 1.49598e+11 m  | au        ,
  Angstrom    | 1e-10 m        | AA, angstrom ,
  Bq          | 1 / s          | becquerel ,
  Ci          | 2.7027e-11 / s | curie     ,
  Hz          | 1 / s          | Hertz, hertz ,
  J           | kg m2 / s2     | Joule, joule ,
  Ry          | 2.17987e-18 kg m2 / s2 | rydberg   ,
  cm          | 0.01 m         | centimeter ,
  eV          | 1.60218e-19 kg m2 / s2 | electronvolt ,
  erg         | 1e-07 kg m2 / s2 |           ,
  k           | 100 / m        | Kayser, kayser ,
  lyr         | 9.46073e+15 m  | lightyear ,
  m           | irreducible    | meter     ,
  micron      | 1e-06 m        |           ,
  pc          | 3.08568e+16 m  | parsec    ,
  solRad      | 6.95508e+08 m  | R_sun, Rsun ,
]
```

Using equivalencies in larger pieces of code

Sometimes one has an involved calculation where one is regularly switching back between equivalent units. For these cases, one can set equivalencies that will by default be used, in a way similar to which one can *enable other units*.

For instance, to enable radian to be treated as a dimensionless unit, simply do:

```
>>> import astropy.units as u
>>> u.set_enabled_equivalencies(u.dimensionless_angles())
<astropy.units.core._UnitContext object at ...>
>>> u.deg.to('')
0.01745329...
```

Here, any list of equivalencies could be used, or one could add, e.g., `spectral()` and `spectral_density()` (since these return lists, they should indeed be combined by adding them together).

The disadvantage of the above approach is that you may forget to turn the default off (done by giving an empty argument). To automate this, a context manager is provided:

```
>>> import astropy.units as u
>>> with u.set_enabled_equivalencies(u.dimensionless_angles()):
...     phase = 0.5 * u.cycle
...     c = np.exp(1j*phase)
>>> c
<Quantity (-1+...j) >
```

6.3.7 Low-level unit conversion

Conversion of quantities from one unit to another is handled using the `Quantity.to` method. This page describes some low-level features for handling unit conversion that are rarely required in user code.

There are two ways of handling conversions between units.

Direct Conversion

In this case, given a source and destination unit, the value(s) in the new units is(are) returned.

```
>>> from astropy import units as u
>>> u.pc.to(u.m, 3.26)
1.0059308915583043e+17
```

This converts 3.26 parsecs to meters.

Arrays are permitted as arguments.

```
>>> u.h.to(u.s, [1, 2, 5, 10.1])
array([ 3600.,  7200., 18000., 36360.]
```

Obtaining a Conversion Function

Finally, one may obtain a function that can be used to convert to the new unit. Normally this may seem like overkill when all one needs to do is multiply by a scale factor, but there are cases when the transformation between units may not be as simple as a single scale factor, for example when a custom equivalency table is in use.

Conversion to different units involves obtaining a conversion function and then applying it to the value, or values to be converted.

```
>>> cms = u.cm / u.s
>>> cms_to_kmph = cms.get_converter(u.km / u.hour)
>>> cms_to_kmph(125.)
4.5
>>> cms_to_kmph([1000, 2000])
array([ 36.,  72.]
```

Incompatible Conversions

If you attempt to convert to a incompatible unit, an exception will result:

```
>>> cms.to(u.km)
Traceback (most recent call last):
...
UnitsError: 'cm / s' (speed) and 'km' (length) are not convertible
Traceback (most recent call last):
...
UnitsError: 'cm / s' (speed) and 'km' (length) are not convertible
```

You can check whether a particular conversion is possible using the `is_equivalent` method:

```
>>> u.m.is_equivalent(u.pc)
True
>>> u.m.is_equivalent("second")
False
>>> (u.m ** 3).is_equivalent(u.l)
True
```

6.4 See Also

- [FITS Standard for units in FITS.](#)
- [The proposed IVOA standard for representing units in the VO.](#)
- [OGIP Units: A standard for storing units in OGIP FITS files.](#)
- [Standards for astronomical catalogues units.](#)
- [IAU Style Manual.](#)
- [A table of astronomical unit equivalencies](#)

6.5 Reference/API

6.5.1 astropy.units.quantity Module

This module defines the `Quantity` object, which represents a number with some associated units. `Quantity` objects support operations like ordinary numbers, but will deal with unit conversions internally.

Classes

`Quantity` A `Quantity` represents a number with some associated unit.

Quantity

class `astropy.units.quantity.Quantity`

Bases: `numpy.ndarray`

A `Quantity` represents a number with some associated unit.

Parameters

value : number, `ndarray`, `Quantity` object, or sequence of `Quantity` objects.

The numerical value of this quantity in the units given by `unit`. If a `Quantity` or sequence of them (or any other valid object with a `unit` attribute), creates a new `Quantity` object, converting to `unit` units as needed.

unit : `UnitBase` instance, str

An object that represents the unit associated with the input value. Must be an `UnitBase` object or a string parseable by the `units` package.

dtype : `~numpy.dtype`, optional

The dtype of the resulting Numpy array or scalar that will hold the value. If not provided, it is determined from the input, except that any input that cannot represent float (integer and bool) is converted to float.

copy : bool, optional

If `True` (default), then the value is copied. Otherwise, a copy will only be made if `__array__` returns a copy, if value is a nested sequence, or if a copy is needed to satisfy an explicitly given `dtype`. (The `False` option is intended mostly for internal use, to speed up initialization where a copy is known to have been made. Use with care.)

order : {'C', 'F', 'A'}, optional

Specify the order of the array. As in `array`. This parameter is ignored if the input is a `Quantity` and `copy=False`.

subok : bool, optional

If `False` (default), the returned array will be forced to be a `Quantity`. Otherwise, `Quantity` subclasses will be passed through.

ndmin : int, optional

Specifies the minimum number of dimensions that the resulting array should have. Ones will be pre-pended to the shape as needed to meet this requirement. This parameter is ignored if the input is a `Quantity` and `copy=False`.

Raises**TypeError**

If the value provided is not a Python numeric type.

TypeError

If the unit provided is not either a `Unit` object or a parseable string unit.

Attributes Summary

<code>cgs</code>	Returns a copy of the current <code>Quantity</code> instance with CGS units.
<code>equivalencies</code>	A list of equivalencies that will be applied by default during unit conversions.
<code>flat</code>	A 1-D iterator over the <code>Quantity</code> array.
<code>isscalar</code>	True if the <code>value</code> of this quantity is a scalar, or False if it is an array-like object.
<code>si</code>	Returns a copy of the current <code>Quantity</code> instance with SI units.
<code>unit</code>	A <code>UnitBase</code> object representing the unit of this quantity.
<code>value</code>	The numerical value of this quantity.

Methods Summary

<code>all([axis, out])</code>	Returns True if all elements evaluate to True.
<code>any([axis, out])</code>	Returns True if any of the elements of a evaluate to True.
<code>argmax([axis, out])</code>	Return indices of the maximum values along the given axis.
<code>argmin([axis, out])</code>	Return indices of the minimum values along the given axis of a.
<code>argsort([axis, kind, order])</code>	Returns the indices that would sort this array.
<code>choose(choices[, out, mode])</code>	Use an index array to construct a new array from a set of choices.
<code>clip(a_min, a_max[, out])</code>	Return an array whose values are limited to <code>[a_min, a_max]</code> .
<code>cumprod([axis, dtype, out])</code>	Return the cumulative product of the elements along the given axis.
<code>cumsum([axis, dtype, out])</code>	Return the cumulative sum of the elements along the given axis.
<code>decompose([bases])</code>	Generates a new <code>Quantity</code> with the units decomposed.
<code>diff([n, axis])</code>	
<code>dot(b[, out])</code>	Dot product of two arrays.
<code>dump(file)</code>	Dump a pickle of the array to the specified file.
<code>dumps()</code>	Returns the pickle of the array as a string.
<code>ediff1d([to_end, to_begin])</code>	
<code>fill(value)</code>	Fill the array with a scalar value.
<code>item(*args)</code>	Copy an element of an array to a standard Python scalar and return it.
<code>itemset(*args)</code>	Insert scalar into an array (scalar is cast to array's dtype, if possible) There must be at least one array in the args.
<code>list()</code>	
<code>max([axis, out])</code>	Return the maximum along a given axis.
<code>mean([axis, dtype, out])</code>	Returns the average of the array elements along given axis.
<code>min([axis, out])</code>	Return the minimum along a given axis.
<code>nansum([axis])</code>	
<code>prod([axis, dtype, out])</code>	Return the product of the array elements over the given axis Refer to <code>numpy.prod</code> for full details.
<code>ptp([axis, out])</code>	Peak to peak (maximum - minimum) value along a given axis.
<code>put(indices, values[, mode])</code>	Set <code>a.flat[n] = values[n]</code> for all <code>n</code> in indices.
<code>searchsorted(v[, side, sorter])</code>	Find indices where elements of <code>v</code> should be inserted in <code>a</code> to maintain order.
<code>std([axis, dtype, out, ddof])</code>	Returns the standard deviation of the array elements along given axis.
<code>sum([axis, dtype, out])</code>	Return the sum of the array elements over the given axis.
<code>to(unit[, equivalencies])</code>	Returns a new <code>Quantity</code> object with the specified units.
<code>tofile(fid[, sep, format])</code>	Write array to a file as text or binary (default).
<code>tostring([order])</code>	Construct Python bytes containing the raw data bytes in the array.
<code>trace([offset, axis1, axis2, dtype, out])</code>	Return the sum along diagonals of the array.
<code>var([axis, dtype, out, ddof])</code>	Returns the variance of the array elements, along given axis.

Attributes Documentation

cgs

Returns a copy of the current `Quantity` instance with CGS units. The value of the resulting object will be scaled.

equivalencies

A list of equivalencies that will be applied by default during unit conversions.

flat

A 1-D iterator over the `Quantity` array.

This returns a `QuantityIterator` instance, which behaves the same as the `flatiter` instance returned by `flat`, and is similar to, but not a subclass of, Python's built-in iterator object.

isscalar

True if the `value` of this quantity is a scalar, or False if it is an array-like object.

Note: This is subtly different from `numpy.isscalar` in that `numpy.isscalar` returns `False` for a zero-dimensional array (e.g. `np.array(1)`), while this is `True` for quantities, since quantities cannot represent true numpy scalars.

si

Returns a copy of the current `Quantity` instance with SI units. The value of the resulting object will be scaled.

unit

A `UnitBase` object representing the unit of this quantity.

value

The numerical value of this quantity.

Methods Documentation

all (*axis=None, out=None*)

Returns `True` if all elements evaluate to `True`.

Refer to `numpy.all` for full documentation.

See also:

`numpy.all`

equivalent function

any (*axis=None, out=None*)

Returns `True` if any of the elements of `a` evaluate to `True`.

Refer to `numpy.any` for full documentation.

See also:

`numpy.any`

equivalent function

argmax (*axis=None, out=None*)

Return indices of the maximum values along the given axis.

Refer to `numpy.argmax` for full documentation.

See also:

`numpy.argmax`

equivalent function

argmin (*axis=None, out=None*)

Return indices of the minimum values along the given axis of `a`.

Refer to `numpy.argmin` for detailed documentation.

See also:

`numpy.argmin`

equivalent function

argsort (*axis=-1, kind='quicksort', order=None*)

Returns the indices that would sort this array.

Refer to `numpy.argsort` for full documentation.

See also:

`numpy.argsort`
equivalent function

choose (*choices, out=None, mode='raise'*)

Use an index array to construct a new array from a set of choices.

Refer to `numpy.choose` for full documentation.

See also:

`numpy.choose`
equivalent function

clip (*a_min, a_max, out=None*)

Return an array whose values are limited to `[a_min, a_max]`.

Refer to `numpy.clip` for full documentation.

See also:

`numpy.clip`
equivalent function

cumprod (*axis=None, dtype=None, out=None*)

Return the cumulative product of the elements along the given axis.

Refer to `numpy.cumprod` for full documentation.

See also:

`numpy.cumprod`
equivalent function

cumsum (*axis=None, dtype=None, out=None*)

Return the cumulative sum of the elements along the given axis.

Refer to `numpy.cumsum` for full documentation.

See also:

`numpy.cumsum`
equivalent function

decompose (*bases=[]*)

Generates a new `Quantity` with the units decomposed. Decomposed units have only irreducible units in them (see `astropy.units.UnitBase.decompose`).

Parameters

bases : sequence of `UnitBase`, optional

The bases to decompose into. When not provided, decomposes down to any irreducible units. When provided, the decomposed result will only contain the given units. This will raise a `UnitsError` if it's not possible to do so.

Returns**newq** : `Quantity`

A new object equal to this quantity with units decomposed.

diff (*n=1, axis=-1*)**dot** (*b, out=None*)

Dot product of two arrays.

Refer to `numpy.dot` for full documentation.**See also:**`numpy.dot`

equivalent function

Examples

```
>>> a = np.eye(2)
>>> b = np.ones((2, 2)) * 2
>>> a.dot(b)
array([[ 2.,  2.],
       [ 2.,  2.]])
```

This array method can be conveniently chained:

```
>>> a.dot(b).dot(b)
array([[ 8.,  8.],
       [ 8.,  8.]])
```

dump (*file*)Dump a pickle of the array to the specified file. The array can be read back with `pickle.load` or `numpy.load`.**Parameters****file** : str

A string naming the dump file.

dumps ()Returns the pickle of the array as a string. `pickle.loads` or `numpy.loads` will convert the string back to an array.**Parameters****None****ediff1d** (*to_end=None, to_begin=None*)**fill** (*value*)

Fill the array with a scalar value.

Parameters**value** : scalarAll elements of `a` will be assigned this value.

Examples

```

>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([ 1.,  1.])

```

`item(*args)`

Copy an element of an array to a standard Python scalar and return it.

Parameters

***args** : Arguments (variable number and type)

- none**: in this case, the method only works for arrays with one element (`a.size == 1`), which element is copied into a standard Python scalar object and returned.
- int_type**: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
- tuple of int_types**: functions as does a single `int_type` argument, except that the argument is interpreted as an nd-index into the array.

Returns

z : Standard Python scalar object

A copy of the specified element of the array as a suitable Python scalar

Notes

When the data type of `a` is `longdouble` or `clongdouble`, `item()` returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for `item()`, unless fields are defined, in which case a tuple is returned.

`item` is very similar to `a[args]`, except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python's optimized math.

Examples

```

>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.item(3)
2
>>> x.item(7)
5
>>> x.item((0, 1))
1
>>> x.item((2, 2))
3

```

itemset (*args)

Insert scalar into an array (scalar is cast to array's dtype, if possible)

There must be at least 1 argument, and define the last argument as *item*. Then, `a.itemset(*args)` is equivalent to but faster than `a[args] = item`. The item should be a scalar value and `args` must select a single item in the array `a`.

Parameters

***args** : Arguments

If one argument: a scalar, only used in case `a` is of size 1. If two arguments: the last argument is the value to be set and must be a scalar, the first argument specifies a single array element location. It is either an int or a tuple.

Notes

Compared to indexing syntax, `itemset` provides some speed increase for placing a scalar into a particular location in an `ndarray`, if you must do this. However, generally this is discouraged: among other problems, it complicates the appearance of the code. Also, when using `itemset` (and `item`) inside a loop, be sure to assign the methods to a local variable to avoid the attribute look-up at each loop iteration.

Examples

```
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[3, 1, 7],
       [2, 0, 3],
       [8, 5, 9]])
```

list ()**max** (axis=None, out=None)

Return the maximum along a given axis.

Refer to `numpy.amax` for full documentation.

See also:**numpy.amax**

equivalent function

mean (axis=None, dtype=None, out=None)

Returns the average of the array elements along given axis.

Refer to `numpy.mean` for full documentation.

See also:**numpy.mean**

equivalent function

min (*axis=None, out=None*)

Return the minimum along a given axis.

Refer to `numpy.amin` for full documentation.

See also:

`numpy.amin`

equivalent function

nansum (*axis=None*)

prod (*axis=None, dtype=None, out=None*)

Return the product of the array elements over the given axis

Refer to `numpy.prod` for full documentation.

See also:

`numpy.prod`

equivalent function

ptp (*axis=None, out=None*)

Peak to peak (maximum - minimum) value along a given axis.

Refer to `numpy.ptp` for full documentation.

See also:

`numpy.ptp`

equivalent function

put (*indices, values, mode='raise'*)

Set `a.flat[n] = values[n]` for all `n` in `indices`.

Refer to `numpy.put` for full documentation.

See also:

`numpy.put`

equivalent function

searchsorted (*v, side='left', sorter=None*)

Find indices where elements of `v` should be inserted in `a` to maintain order.

For full documentation, see `numpy.searchsorted`

See also:

`numpy.searchsorted`

equivalent function

std (*axis=None, dtype=None, out=None, ddof=0*)

Returns the standard deviation of the array elements along given axis.

Refer to `numpy.std` for full documentation.

See also:

`numpy.std`
equivalent function

sum (*axis=None, dtype=None, out=None*)
Return the sum of the array elements over the given axis.

Refer to `numpy.sum` for full documentation.

See also:

`numpy.sum`
equivalent function

to (*unit, equivalencies=[]*)
Returns a new `Quantity` object with the specified units.

Parameters

unit : `UnitBase` instance, str

An object that represents the unit to convert to. Must be an `UnitBase` object or a string parseable by the `units` package.

equivalencies : list of equivalence pairs, optional

A list of equivalence pairs to try if the units are not directly convertible. See *Equivalencies*. If not provided or [], class default equivalencies will be used (none for `Quantity`, but may be set for subclasses) If `None`, no equivalencies will be applied at all, not even any set globally or within a context.

tofile (*fid, sep=" ", format="%s"*)
Write array to a file as text or binary (default).

Data is always written in 'C' order, independent of the order of `a`. The data produced by this method can be recovered using the function `fromfile()`.

Parameters

fid : file or str

An open file object, or a string containing a filename.

sep : str

Separator between array items for text output. If "" (empty), a binary file is written, equivalent to `file.write(a.tobytes())`.

format : str

Format string for text file output. Each entry in the array is formatted to text by first converting it to the closest Python type, and then using "format" % item.

Notes

This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

tostring (*order='C'*)
Construct Python bytes containing the raw data bytes in the array.

Constructs Python bytes showing a copy of the raw contents of data memory. The bytes object can be produced in either 'C' or 'Fortran', or 'Any' order (the default is 'C'-order). 'Any' order means C-order unless the `F_CONTIGUOUS` flag in the array is set, in which case it means 'Fortran' order.

This function is a compatibility alias for `tobytes`. Despite its name it returns bytes not strings.

Parameters

order : {'C', 'F', None}, optional

Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

Returns

s : bytes

Python bytes exhibiting a copy of a's raw data.

Examples

```
>>> x = np.array([[0, 1], [2, 3]])
>>> x.tobytes()
b'\x00\x00\x00\x00\x01\x00\x00\x00\x02\x00\x00\x00\x03\x00\x00\x00'
>>> x.tobytes('C') == x.tobytes()
True
>>> x.tobytes('F')
b'\x00\x00\x00\x00\x02\x00\x00\x00\x01\x00\x00\x00\x03\x00\x00\x00'
```

trace (*offset=0, axis1=0, axis2=1, dtype=None, out=None*)

Return the sum along diagonals of the array.

Refer to `numpy.trace` for full documentation.

See also:

`numpy.trace`

equivalent function

var (*axis=None, dtype=None, out=None, ddof=0*)

Returns the variance of the array elements, along given axis.

Refer to `numpy.var` for full documentation.

See also:

`numpy.var`

equivalent function

Class Inheritance Diagram

6.5.2 astropy.units Module

This subpackage contains classes and functions for defining and converting between different physical units.

This code is adapted from the `pynbody` units module written by Andrew Pontzen, who has granted the Astropy project permission to use the code under a BSD license.

Functions

<code>add_enabled_equivalencies(equivalencies)</code>	Adds to the equivalencies enabled in the unit registry.
<code>add_enabled_units(units)</code>	Adds to the set of units enabled in the unit registry.
<code>brightness_temperature(beam_area, disp)</code>	Defines the conversion between Jy/beam and “brightness temperature”, T_B , in
<code>def_physical_type(unit, name)</code>	Adds a new physical unit mapping.
<code>def_unit(s[, represents, register, doc, ...])</code>	Factory function for defining new units.
<code>dimensionless_angles()</code>	Allow angles to be equivalent to dimensionless (with 1 rad = 1 m/m = 1).
<code>doppler_optical(rest)</code>	Return the equivalency pairs for the optical convention for velocity.
<code>doppler_radio(rest)</code>	Return the equivalency pairs for the radio convention for velocity.
<code>doppler_relativistic(rest)</code>	Return the equivalency pairs for the relativistic convention for velocity.
<code>get_current_unit_registry()</code>	
<code>get_physical_type(unit)</code>	Given a unit, returns the name of the physical quantity it represents.
<code>logarithmic()</code>	Allow logarithmic units to be converted to dimensionless fractions
<code>mass_energy()</code>	Returns a list of equivalence pairs that handle the conversion between mass and
<code>parallax()</code>	Returns a list of equivalence pairs that handle the conversion between parallax and
<code>set_enabled_equivalencies(equivalencies)</code>	Sets the equivalencies enabled in the unit registry.
<code>set_enabled_units(units)</code>	Sets the units enabled in the unit registry.
<code>spectral()</code>	Returns a list of equivalence pairs that handle spectral wavelength, wave number
<code>spectral_density(wav[, factor])</code>	Returns a list of equivalence pairs that handle spectral density with regard to w
<code>temperature()</code>	Convert between Kelvin, Celsius, and Fahrenheit here because Unit and Comp
<code>temperature_energy()</code>	Convert between Kelvin and keV(eV) to an equivalent amount.

add_enabled_equivalencies

`astropy.units.add_enabled_equivalencies` (*equivalencies*)

Adds to the equivalencies enabled in the unit registry.

These equivalencies are used if no explicit equivalencies are given, both in unit conversion and in finding equivalent units.

This is meant in particular for allowing angles to be dimensionless. Since no equivalencies are enabled by default, generally it is recommended to use `set_enabled_equivalencies`.

Parameters

equivalencies : list of equivalent pairs

E.g., as returned by `dimensionless_angles`.

add_enabled_units

`astropy.units.add_enabled_units` (*units*)

Adds to the set of units enabled in the unit registry.

These units are searched when using `UnitBase.find_equivalent_units`, for example.

This may be used either permanently, or as a context manager using the `with` statement (see example below).

Parameters

units : list of sequences, dicts, or modules containing units, or units

This is a list of things in which units may be found (sequences, dicts or modules), or units themselves. The entire set will be added to the “enabled” set for searching through by methods like `UnitBase.find_equivalent_units` and `UnitBase.compose`.

Examples

```
>>> from astropy import units as u
>>> from astropy.units import imperial
>>> with u.add_enabled_units(imperial):
...     u.m.find_equivalent_units()
...
Primary name | Unit definition | Aliases
[
  AU          | 1.49598e+11 m   | au
  Angstrom    | 1e-10 m         | AA, angstrom
  cm          | 0.01 m          | centimeter
  ft          | 0.3048 m        | foot
  inch        | 0.0254 m        |
  lyr         | 9.46073e+15 m   | lightyear
  m           | irreducible     | meter
  mi          | 1609.34 m       | mile
  micron      | 1e-06 m         |
  nmi         | 1852 m          | nauticalmile, NM
  pc          | 3.08568e+16 m   | parsec
  solRad      | 6.95508e+08 m   | R_sun, Rsun
  yd          | 0.9144 m        | yard
]
```

brightness_temperature

`astropy.units.brightness_temperature` (*beam_area, disp*)

Defines the conversion between Jy/beam and “brightness temperature”, T_B , in Kelvins. The brightness temperature is a unit very commonly used in radio astronomy. See, e.g., “Tools of Radio Astronomy” (Wilson 2009) eqn 8.16 and eqn 8.19 (these pages are available on [google books](#)).

$$T_B \equiv S_\nu / (2k\nu^2/c^2)$$

However, the beam area is essential for this computation: the brightness temperature is inversely proportional to the beam area

Parameters

beam_area : Beam Area equivalent

Beam area in angular units, i.e. steradian equivalent

disp : `Quantity` with spectral units

The observed `spectral` equivalent `Unit` (e.g., frequency or wavelength)

Examples

Arecibo C-band beam:

```
>>> import numpy as np
>>> from astropy import units as u
>>> beam_area = np.pi*(50*u.arcsec)**2
>>> freq = 5*u.GHz
>>> equiv = u.brightness_temperature(beam_area, freq)
>>> u.Jy.to(u.K, equivalencies=equiv)
7.052588858846446
```

```
>>> (1*u.Jy).to(u.K, equivalencies=equiv)
<Quantity 7.052588858846446 K>
```

VLA synthetic beam:

```
>>> beam_area = np.pi*(15*u.arcsec)**2
>>> freq = 5*u.GHz
>>> equiv = u.brightness_temperature(beam_area, freq)
>>> u.Jy.to(u.K, equivalencies=equiv)
78.36209843162719
```

def_physical_type

`astropy.units.def_physical_type` (*unit*, *name*)
Adds a new physical unit mapping.

Parameters

unit : `UnitBase` instance

The unit to map from.

name : str

The physical name of the unit.

def_unit

`astropy.units.def_unit` (*s*, *represents=None*, *register=None*, *doc=None*, *format=None*, *prefixes=False*, *exclude_prefixes=[]*, *namespace=None*)
Factory function for defining new units.

Parameters

s : str or list of str

The name of the unit. If a list, the first element is the canonical (short) name, and the rest of the elements are aliases.

represents : `UnitBase` instance, optional

The unit that this named unit represents. If not provided, a new `IrreducibleUnit` is created.

doc : str, optional

A docstring describing the unit.

format : dict, optional

A mapping to format-specific representations of this unit. For example, for the `Ohm` unit, it might be nice to have it displayed as `\Omega` by the `latex` formatter. In that case, `format` argument should be set to:

```
{ 'latex': r'\Omega' }
```

prefixes : bool or list, optional

When `True`, generate all of the SI prefixed versions of the unit as well. For example, for a given unit `m`, will generate `mm`, `cm`, `km`, etc. When a list, it is a list of prefix definitions of the form:

(short_names, long_tables, factor)

Default is `False`. This function always returns the base unit object, even if multiple scaled versions of the unit were created.

exclude_prefixes : list of str, optional

If any of the SI prefixes need to be excluded, they may be listed here. For example, Pa can be interpreted either as “petaannum” or “Pascal”. Therefore, when defining the prefixes for a, `exclude_prefixes` should be set to `["P"]`.

namespace : dict, optional

When provided, inject the unit (and all of its aliases and prefixes), into the given namespace dictionary.

Returns

unit : `UnitBase` object

The newly-defined unit, or a matching unit that was already defined.

dimensionless_angles

`astropy.units.dimensionless_angles()`

Allow angles to be equivalent to dimensionless (with $1 \text{ rad} = 1 \text{ m/m} = 1$).

It is special compared to other equivalency pairs in that it allows this independent of the power to which the angle is raised, and independent of whether it is part of a more complicated unit.

doppler_optical

`astropy.units.doppler_optical(rest)`

Return the equivalency pairs for the optical convention for velocity.

The optical convention for the relation between velocity and frequency is:

$$V = c \frac{f_0 - f}{f}; f(V) = f_0(1 + V/c)^{-1}$$

Parameters

rest : `Quantity`

Any quantity supported by the standard spectral equivalencies (wavelength, energy, frequency, wave number).

References

[NRAO site defining the conventions](#)

Examples

```
>>> import astropy.units as u
>>> CO_restfreq = 115.27120*u.GHz # rest frequency of 12 CO 1-0 in GHz
>>> optical_CO_equiv = u.doppler_optical(CO_restfreq)
>>> measured_freq = 115.2832*u.GHz
>>> optical_velocity = measured_freq.to(u.km/u.s, equivalencies=optical_CO_equiv)
>>> optical_velocity
<Quantity -31.20584348799674 km / s>
```

doppler_radio

`astropy.units.doppler_radio` (*rest*)

Return the equivalency pairs for the radio convention for velocity.

The radio convention for the relation between velocity and frequency is:

$$V = c \frac{f_0 - f}{f_0}; f(V) = f_0(1 - V/c)$$

Parameters

rest: `Quantity`

Any quantity supported by the standard spectral equivalencies (wavelength, energy, frequency, wave number).

References

[NRAO site defining the conventions](#)

Examples

```
>>> import astropy.units as u
>>> CO_restfreq = 115.27120*u.GHz # rest frequency of 12 CO 1-0 in GHz
>>> radio_CO_equiv = u.doppler_radio(CO_restfreq)
>>> measured_freq = 115.2832*u.GHz
>>> radio_velocity = measured_freq.to(u.km/u.s, equivalencies=radio_CO_equiv)
>>> radio_velocity
<Quantity -31.209092088877583 km / s>
```

doppler_relativistic

`astropy.units.doppler_relativistic` (*rest*)

Return the equivalency pairs for the relativistic convention for velocity.

The full relativistic convention for the relation between velocity and frequency is:

$$V = c \frac{f_0^2 - f^2}{f_0^2 + f^2}; f(V) = f_0 \frac{(1 - (V/c)^2)^{1/2}}{(1 + V/c)}$$

Parameters

rest: `Quantity`

Any quantity supported by the standard spectral equivalencies (wavelength, energy, frequency, wave number).

References

[NRAO site defining the conventions](#)

Examples

```

>>> import astropy.units as u
>>> CO_restfreq = 115.27120*u.GHz # rest frequency of 12 CO 1-0 in GHz
>>> relativistic_CO_equiv = u.doppler_relativistic(CO_restfreq)
>>> measured_freq = 115.2832*u.GHz
>>> relativistic_velocity = measured_freq.to(u.km/u.s, equivalencies=relativistic_CO_equiv)
>>> relativistic_velocity
<Quantity -31.207467619351537 km / s>
>>> measured_velocity = 1250 * u.km/u.s
>>> relativistic_frequency = measured_velocity.to(u.GHz, equivalencies=relativistic_CO_equiv)
>>> relativistic_frequency
<Quantity 114.79156866993588 GHz>
>>> relativistic_wavelength = measured_velocity.to(u.mm, equivalencies=relativistic_CO_equiv)
>>> relativistic_wavelength
<Quantity 2.6116243681798923 mm>

```

get_current_unit_registry

```
astropy.units.get_current_unit_registry()
```

get_physical_type

```
astropy.units.get_physical_type(unit)
```

Given a unit, returns the name of the physical quantity it represents. If it represents an unknown physical quantity, "unknown" is returned.

Parameters

unit: `UnitBase` instance

The unit to lookup

Returns

physical: str

The name of the physical quantity, or unknown if not known.

logarithmic

```
astropy.units.logarithmic()
```

Allow logarithmic units to be converted to dimensionless fractions

mass_energy

```
astropy.units.mass_energy()
```

Returns a list of equivalence pairs that handle the conversion between mass and energy.

parallax

```
astropy.units.parallax()
```

Returns a list of equivalence pairs that handle the conversion between parallax angle and distance.

set_enabled_equivalencies

`astropy.units.set_enabled_equivalencies` (*equivalencies*)

Sets the equivalencies enabled in the unit registry.

These equivalencies are used if no explicit equivalencies are given, both in unit conversion and in finding equivalent units.

This is meant in particular for allowing angles to be dimensionless. Use with care.

Parameters

equivalencies : list of equivalent pairs

E.g., as returned by `dimensionless_angles`.

Examples

Exponentiation normally requires dimensionless quantities. To avoid problems with complex phases:

```
>>> from astropy import units as u
>>> with u.set_enabled_equivalencies(u.dimensionless_angles()):
...     phase = 0.5 * u.cycle
...     np.exp(1j*phase)
<Quantity (-1+1.2246063538223773e-16j)>
```

set_enabled_units

`astropy.units.set_enabled_units` (*units*)

Sets the units enabled in the unit registry.

These units are searched when using `UnitBase.find_equivalent_units`, for example.

This may be used either permanently, or as a context manager using the `with` statement (see example below).

Parameters

units : list of sequences, dicts, or modules containing units, or units

This is a list of things in which units may be found (sequences, dicts or modules), or units themselves. The entire set will be “enabled” for searching through by methods like `UnitBase.find_equivalent_units` and `UnitBase.compose`.

Examples

```
>>> from astropy import units as u
>>> with u.set_enabled_units([u.pc]):
...     u.m.find_equivalent_units()
...
Primary name | Unit definition | Aliases
[
  pc          | 3.08568e+16 m   | parsec ,
]
>>> u.m.find_equivalent_units()
Primary name | Unit definition | Aliases
[
  AU          | 1.49598e+11 m   | au          ,
  Angstrom    | 1e-10 m         | AA, angstrom ,
```

```

cm          | 0.01 m          | centimeter  ,
lyr         | 9.46073e+15 m  | lightyear   ,
m           | irreducible    | meter       ,
micron      | 1e-06 m        |             ,
pc          | 3.08568e+16 m  | parsec      ,
solRad      | 6.95508e+08 m  | R_sun, R_sun ,
]

```

spectral

`astropy.units.spectral()`

Returns a list of equivalence pairs that handle spectral wavelength, wave number, frequency, and energy equivalences.

Allows conversions between wavelength units, wave number units, frequency units, and energy units as they relate to light.

There are two types of wave number:

- spectroscopic - $1/\lambda$ (per meter)
- angular - $2\pi/\lambda$ (radian per meter)

spectral_density

`astropy.units.spectral_density(wav, factor=None)`

Returns a list of equivalence pairs that handle spectral density with regard to wavelength and frequency.

Parameters

wav: `Quantity`

`Quantity` associated with values being converted (e.g., wavelength or frequency).

Notes

The `factor` argument is left for backward-compatibility with the syntax `spectral_density(unit, factor)` but users are encouraged to use `spectral_density(factor * unit)` instead.

temperature

`astropy.units.temperature()`

Convert between Kelvin, Celsius, and Fahrenheit here because `Unit` and `CompositeUnit` cannot do addition or subtraction properly.

temperature_energy

`astropy.units.temperature_energy()`

Convert between Kelvin and keV(eV) to an equivalent amount.

Classes

<code>CompositeUnit(scale, bases, powers[, ...])</code>	Create a composite unit using expressions of previously defined units.
<code>IrreducibleUnit(st[, register, doc, format, ...])</code>	Irreducible units are the units that all other units are defined in terms of.
<code>NamedUnit(st[, register, doc, format, namespace])</code>	The base class of units that have a name.
<code>PrefixUnit(st[, represents, register, doc, ...])</code>	A unit that is simply a SI-prefixed version of another unit.
<code>Quantity</code>	A <code>Quantity</code> represents a number with some associated unit.
<code>Unit(st[, represents, register, doc, ...])</code>	The main unit class.
<code>UnitBase</code>	Abstract base class for units.
<code>UnitsError</code>	The base class for unit-specific exceptions.
<code>UnitsWarning</code>	The base class for unit-specific exceptions.
<code>UnrecognizedUnit(st[, register, doc, ...])</code>	A unit that did not parse correctly.

CompositeUnit

```
class astropy.units.CompositeUnit (scale, bases, powers, decompose=False, decom-  
pose_bases=set([], _error_check=True)
```

Bases: `astropy.units.UnitBase`

Create a composite unit using expressions of previously defined units.

Direct use of this class is not recommended. Instead use the factory function `Unit` and arithmetic operators to compose units.

Parameters

scale : number

A scaling factor for the unit.

bases : sequence of `UnitBase`

A sequence of units this unit is composed of.

powers : sequence of numbers

A sequence of powers (in parallel with `bases`) for each of the base units.

Attributes Summary

<code>bases</code>	Return the bases of the composite unit.
<code>powers</code>	Return the powers of the composite unit.
<code>scale</code>	Return the scale of the composite unit.

Methods Summary

<code>decompose([bases])</code>	Return a unit object composed of only irreducible units.
<code>is_unity()</code>	Returns <code>True</code> if the unit is unscaled and dimensionless.

Attributes Documentation

bases

Return the bases of the composite unit.

powers

Return the powers of the composite unit.

scale

Return the scale of the composite unit.

Methods Documentation**decompose** (*bases=set([])*)

Return a unit object composed of only irreducible units.

Parameters

bases : sequence of UnitBase, optional

The bases to decompose into. When not provided, decomposes down to any irreducible units. When provided, the decomposed result will only contain the given units. This will raise a `UnitsError` if it's not possible to do so.

Returns

unit : CompositeUnit object

New object containing only irreducible unit objects.

is_unity ()

Returns `True` if the unit is unscaled and dimensionless.

IrreducibleUnit

```
class astropy.units.IrreducibleUnit (st, register=None, doc=None, format=None, namespace=None)
```

Bases: `astropy.units.NamedUnit`

Irreducible units are the units that all other units are defined in terms of.

Examples are meters, seconds, kilograms, amperes, etc. There is only once instance of such a unit per type.

Methods Summary

`decompose([bases])` Return a unit object composed of only irreducible units.

Methods Documentation**decompose** (*bases=set([])*)

Return a unit object composed of only irreducible units.

Parameters

bases : sequence of UnitBase, optional

The bases to decompose into. When not provided, decomposes down to any irreducible units. When provided, the decomposed result will only contain the given units. This will raise a `UnitsError` if it's not possible to do so.

Returns

unit : CompositeUnit object

New object containing only irreducible unit objects.

NamedUnit

class `astropy.units.NamedUnit` (*st*, *register=None*, *doc=None*, *format=None*, *namespace=None*)

Bases: `astropy.units.UnitBase`

The base class of units that have a name.

Parameters

st : str, list of str, 2-tuple

The name of the unit. If a list of strings, the first element is the canonical (short) name, and the rest of the elements are aliases. If a tuple of lists, the first element is a list of short names, and the second element is a list of long names; all but the first short name are considered “aliases”. Each name *should* be a valid Python identifier to make it easy to access, but this is not required.

namespace : dict, optional

When provided, inject the unit, and all of its aliases, in the given namespace dictionary. If a unit by the same name is already in the namespace, a `ValueError` is raised.

doc : str, optional

A docstring describing the unit.

format : dict, optional

A mapping to format-specific representations of this unit. For example, for the Ohm unit, it might be nice to have it displayed as Ω by the `latex` formatter. In that case, `format` argument should be set to:

```
{'latex': r'\Omega'}
```

Raises

ValueError

If any of the given unit names are already in the registry.

ValueError

If any of the given unit names are not valid Python tokens.

Attributes Summary

<code>aliases</code>	Returns the alias (long) names for this unit.
<code>long_names</code>	Returns all of the long names associated with this unit.
<code>name</code>	Returns the canonical (short) name associated with this unit.
<code>names</code>	Returns all of the names associated with this unit.
<code>short_names</code>	Returns all of the short names associated with this unit.

Methods Summary

<code>deregister([remove_from_namespace])</code>	
<code>get_format_name(format)</code>	Get a name for this unit that is specific to a particular format.
<code>register([add_to_namespace])</code>	

Attributes Documentation

aliases

Returns the alias (long) names for this unit.

long_names

Returns all of the long names associated with this unit.

name

Returns the canonical (short) name associated with this unit.

names

Returns all of the names associated with this unit.

short_names

Returns all of the short names associated with this unit.

Methods Documentation

deregister (*remove_from_namespace=False*)

get_format_name (*format*)

Get a name for this unit that is specific to a particular format.

Uses the dictionary passed into the `format` kwarg in the constructor.

Parameters

format : str

The name of the format

Returns

name : str

The name of the unit for the given format.

register (*add_to_namespace=False*)

PrefixUnit

class `astropy.units.PrefixUnit` (*st*, *represents=None*, *register=None*, *doc=None*, *format=None*,
namespace=None)

Bases: `astropy.units.Unit`

A unit that is simply a SI-prefixed version of another unit.

For example, mm is a `PrefixUnit` of `.001 * m`.

The constructor is the same as for `Unit`.

Quantity

class `astropy.units.Quantity`

Bases: `numpy.ndarray`

A `Quantity` represents a number with some associated unit.

Parameters

value : number, `ndarray`, `Quantity` object, or sequence of `Quantity` objects.

The numerical value of this quantity in the units given by `unit`. If a `Quantity` or sequence of them (or any other valid object with a `unit` attribute), creates a new `Quantity` object, converting to `unit` units as needed.

unit : `UnitBase` instance, str

An object that represents the unit associated with the input value. Must be an `UnitBase` object or a string parseable by the `units` package.

dtype : `~numpy.dtype`, optional

The dtype of the resulting Numpy array or scalar that will hold the value. If not provided, it is determined from the input, except that any input that cannot represent float (integer and bool) is converted to float.

copy : bool, optional

If `True` (default), then the value is copied. Otherwise, a copy will only be made if `__array__` returns a copy, if value is a nested sequence, or if a copy is needed to satisfy an explicitly given `dtype`. (The `False` option is intended mostly for internal use, to speed up initialization where a copy is known to have been made. Use with care.)

order : {'C', 'F', 'A'}, optional

Specify the order of the array. As in `array`. This parameter is ignored if the input is a `Quantity` and `copy=False`.

subok : bool, optional

If `False` (default), the returned array will be forced to be a `Quantity`. Otherwise, `Quantity` subclasses will be passed through.

ndmin : int, optional

Specifies the minimum number of dimensions that the resulting array should have. Ones will be pre-pended to the shape as needed to meet this requirement. This parameter is ignored if the input is a `Quantity` and `copy=False`.

Raises**TypeError**

If the value provided is not a Python numeric type.

TypeError

If the unit provided is not either a `Unit` object or a parseable string unit.

Attributes Summary

<code>cgs</code>	Returns a copy of the current <code>Quantity</code> instance with CGS units.
<code>equivalencies</code>	A list of equivalencies that will be applied by default during unit conversions.
<code>flat</code>	A 1-D iterator over the <code>Quantity</code> array.
<code>isscalar</code>	True if the <code>value</code> of this quantity is a scalar, or False if it is an array-like object.
<code>si</code>	Returns a copy of the current <code>Quantity</code> instance with SI units.
<code>unit</code>	A <code>UnitBase</code> object representing the unit of this quantity.
<code>value</code>	The numerical value of this quantity.

Methods Summary

<code>all([axis, out])</code>	Returns True if all elements evaluate to True.
<code>any([axis, out])</code>	Returns True if any of the elements of a evaluate to True.
<code>argmax([axis, out])</code>	Return indices of the maximum values along the given axis.
<code>argmin([axis, out])</code>	Return indices of the minimum values along the given axis of a.
<code>argsort([axis, kind, order])</code>	Returns the indices that would sort this array.
<code>choose(choices[, out, mode])</code>	Use an index array to construct a new array from a set of choices.
<code>clip(a_min, a_max[, out])</code>	Return an array whose values are limited to <code>[a_min, a_max]</code> .
<code>cumprod([axis, dtype, out])</code>	Return the cumulative product of the elements along the given axis.
<code>cumsum([axis, dtype, out])</code>	Return the cumulative sum of the elements along the given axis.
<code>decompose([bases])</code>	Generates a new <code>Quantity</code> with the units decomposed.
<code>diff([n, axis])</code>	
<code>dot(b[, out])</code>	Dot product of two arrays.
<code>dump(file)</code>	Dump a pickle of the array to the specified file.
<code>dumps()</code>	Returns the pickle of the array as a string.
<code>ediff1d([to_end, to_begin])</code>	
<code>fill(value)</code>	Fill the array with a scalar value.
<code>item(*args)</code>	Copy an element of an array to a standard Python scalar and return it.
<code>itemset(*args)</code>	Insert scalar into an array (scalar is cast to array's dtype, if possible) There must be at least one array in the args.
<code>list()</code>	
<code>max([axis, out])</code>	Return the maximum along a given axis.
<code>mean([axis, dtype, out])</code>	Returns the average of the array elements along given axis.
<code>min([axis, out])</code>	Return the minimum along a given axis.
<code>nansum([axis])</code>	
<code>prod([axis, dtype, out])</code>	Return the product of the array elements over the given axis Refer to <code>numpy.prod</code> for full details.
<code>ptp([axis, out])</code>	Peak to peak (maximum - minimum) value along a given axis.
<code>put(indices, values[, mode])</code>	Set <code>a.flat[n] = values[n]</code> for all <code>n</code> in indices.
<code>searchsorted(v[, side, sorter])</code>	Find indices where elements of <code>v</code> should be inserted in <code>a</code> to maintain order.
<code>std([axis, dtype, out, ddof])</code>	Returns the standard deviation of the array elements along given axis.
<code>sum([axis, dtype, out])</code>	Return the sum of the array elements over the given axis.
<code>to(unit[, equivalencies])</code>	Returns a new <code>Quantity</code> object with the specified units.
<code>tofile(fid[, sep, format])</code>	Write array to a file as text or binary (default).
<code>tostring([order])</code>	Construct Python bytes containing the raw data bytes in the array.
<code>trace([offset, axis1, axis2, dtype, out])</code>	Return the sum along diagonals of the array.
<code>var([axis, dtype, out, ddof])</code>	Returns the variance of the array elements, along given axis.

Attributes Documentation

cgs

Returns a copy of the current `Quantity` instance with CGS units. The value of the resulting object will be scaled.

equivalencies

A list of equivalencies that will be applied by default during unit conversions.

flat

A 1-D iterator over the `Quantity` array.

This returns a `QuantityIterator` instance, which behaves the same as the `flatiter` instance returned by `flat`, and is similar to, but not a subclass of, Python's built-in iterator object.

isscalar

True if the `value` of this quantity is a scalar, or False if it is an array-like object.

Note: This is subtly different from `numpy.isscalar` in that `numpy.isscalar` returns `False` for a zero-dimensional array (e.g. `np.array(1)`), while this is `True` for quantities, since quantities cannot represent true numpy scalars.

si

Returns a copy of the current `Quantity` instance with SI units. The value of the resulting object will be scaled.

unit

A `UnitBase` object representing the unit of this quantity.

value

The numerical value of this quantity.

Methods Documentation

all (*axis=None, out=None*)

Returns `True` if all elements evaluate to `True`.

Refer to `numpy.all` for full documentation.

See also:

`numpy.all`

equivalent function

any (*axis=None, out=None*)

Returns `True` if any of the elements of `a` evaluate to `True`.

Refer to `numpy.any` for full documentation.

See also:

`numpy.any`

equivalent function

argmax (*axis=None, out=None*)

Return indices of the maximum values along the given axis.

Refer to `numpy.argmax` for full documentation.

See also:

`numpy.argmax`

equivalent function

argmin (*axis=None, out=None*)

Return indices of the minimum values along the given axis of `a`.

Refer to `numpy.argmin` for detailed documentation.

See also:

`numpy.argmin`

equivalent function

argsort (*axis=-1, kind='quicksort', order=None*)

Returns the indices that would sort this array.

Refer to `numpy.argsort` for full documentation.

See also:

`numpy.argsort`
equivalent function

choose (*choices, out=None, mode='raise'*)

Use an index array to construct a new array from a set of choices.

Refer to `numpy.choose` for full documentation.

See also:

`numpy.choose`
equivalent function

clip (*a_min, a_max, out=None*)

Return an array whose values are limited to `[a_min, a_max]`.

Refer to `numpy.clip` for full documentation.

See also:

`numpy.clip`
equivalent function

cumprod (*axis=None, dtype=None, out=None*)

Return the cumulative product of the elements along the given axis.

Refer to `numpy.cumprod` for full documentation.

See also:

`numpy.cumprod`
equivalent function

cumsum (*axis=None, dtype=None, out=None*)

Return the cumulative sum of the elements along the given axis.

Refer to `numpy.cumsum` for full documentation.

See also:

`numpy.cumsum`
equivalent function

decompose (*bases=[]*)

Generates a new `Quantity` with the units decomposed. Decomposed units have only irreducible units in them (see `astropy.units.UnitBase.decompose`).

Parameters

bases : sequence of `UnitBase`, optional

The bases to decompose into. When not provided, decomposes down to any irreducible units. When provided, the decomposed result will only contain the given units. This will raise a `UnitsError` if it's not possible to do so.

Returns**newq** : `Quantity`

A new object equal to this quantity with units decomposed.

diff (*n=1, axis=-1*)**dot** (*b, out=None*)

Dot product of two arrays.

Refer to `numpy.dot` for full documentation.**See also:**`numpy.dot`

equivalent function

Examples

```
>>> a = np.eye(2)
>>> b = np.ones((2, 2)) * 2
>>> a.dot(b)
array([[ 2.,  2.],
       [ 2.,  2.]])
```

This array method can be conveniently chained:

```
>>> a.dot(b).dot(b)
array([[ 8.,  8.],
       [ 8.,  8.]])
```

dump (*file*)Dump a pickle of the array to the specified file. The array can be read back with `pickle.load` or `numpy.load`.**Parameters****file** : str

A string naming the dump file.

dumps ()Returns the pickle of the array as a string. `pickle.loads` or `numpy.loads` will convert the string back to an array.**Parameters****None****ediff1d** (*to_end=None, to_begin=None*)**fill** (*value*)

Fill the array with a scalar value.

Parameters**value** : scalarAll elements of `a` will be assigned this value.

Examples

```

>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([ 1.,  1.])

```

`item(*args)`

Copy an element of an array to a standard Python scalar and return it.

Parameters

***args** : Arguments (variable number and type)

- none**: in this case, the method only works for arrays with one element (`a.size == 1`), which element is copied into a standard Python scalar object and returned.
- int_type**: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
- tuple of int_types**: functions as does a single `int_type` argument, except that the argument is interpreted as an nd-index into the array.

Returns

z : Standard Python scalar object

A copy of the specified element of the array as a suitable Python scalar

Notes

When the data type of `a` is `longdouble` or `clongdouble`, `item()` returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for `item()`, unless fields are defined, in which case a tuple is returned.

`item` is very similar to `a[args]`, except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python's optimized math.

Examples

```

>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.item(3)
2
>>> x.item(7)
5
>>> x.item((0, 1))
1
>>> x.item((2, 2))
3

```

itemset (*args)

Insert scalar into an array (scalar is cast to array's dtype, if possible)

There must be at least 1 argument, and define the last argument as *item*. Then, `a.itemset(*args)` is equivalent to but faster than `a[args] = item`. The item should be a scalar value and `args` must select a single item in the array `a`.

Parameters

***args** : Arguments

If one argument: a scalar, only used in case `a` is of size 1. If two arguments: the last argument is the value to be set and must be a scalar, the first argument specifies a single array element location. It is either an int or a tuple.

Notes

Compared to indexing syntax, `itemset` provides some speed increase for placing a scalar into a particular location in an `ndarray`, if you must do this. However, generally this is discouraged: among other problems, it complicates the appearance of the code. Also, when using `itemset` (and `item`) inside a loop, be sure to assign the methods to a local variable to avoid the attribute look-up at each loop iteration.

Examples

```
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[3, 1, 7],
       [2, 0, 3],
       [8, 5, 9]])
```

list ()**max** (axis=None, out=None)

Return the maximum along a given axis.

Refer to `numpy.amax` for full documentation.

See also:**numpy.amax**

equivalent function

mean (axis=None, dtype=None, out=None)

Returns the average of the array elements along given axis.

Refer to `numpy.mean` for full documentation.

See also:**numpy.mean**

equivalent function

min (*axis=None, out=None*)

Return the minimum along a given axis.

Refer to `numpy.amin` for full documentation.

See also:

`numpy.amin`

equivalent function

nansum (*axis=None*)

prod (*axis=None, dtype=None, out=None*)

Return the product of the array elements over the given axis

Refer to `numpy.prod` for full documentation.

See also:

`numpy.prod`

equivalent function

ptp (*axis=None, out=None*)

Peak to peak (maximum - minimum) value along a given axis.

Refer to `numpy.ptp` for full documentation.

See also:

`numpy.ptp`

equivalent function

put (*indices, values, mode='raise'*)

Set `a.flat[n] = values[n]` for all `n` in `indices`.

Refer to `numpy.put` for full documentation.

See also:

`numpy.put`

equivalent function

searchsorted (*v, side='left', sorter=None*)

Find indices where elements of `v` should be inserted in `a` to maintain order.

For full documentation, see `numpy.searchsorted`

See also:

`numpy.searchsorted`

equivalent function

std (*axis=None, dtype=None, out=None, ddof=0*)

Returns the standard deviation of the array elements along given axis.

Refer to `numpy.std` for full documentation.

See also:

`numpy.std`
equivalent function

sum (*axis=None, dtype=None, out=None*)
Return the sum of the array elements over the given axis.

Refer to `numpy.sum` for full documentation.

See also:

`numpy.sum`
equivalent function

to (*unit, equivalencies=[]*)
Returns a new `Quantity` object with the specified units.

Parameters

unit : `UnitBase` instance, str

An object that represents the unit to convert to. Must be an `UnitBase` object or a string parseable by the `units` package.

equivalencies : list of equivalence pairs, optional

A list of equivalence pairs to try if the units are not directly convertible. See *Equivalencies*. If not provided or [], class default equivalencies will be used (none for `Quantity`, but may be set for subclasses) If `None`, no equivalencies will be applied at all, not even any set globally or within a context.

tofile (*fid, sep=" ", format="%s"*)
Write array to a file as text or binary (default).

Data is always written in 'C' order, independent of the order of `a`. The data produced by this method can be recovered using the function `fromfile()`.

Parameters

fid : file or str

An open file object, or a string containing a filename.

sep : str

Separator between array items for text output. If "" (empty), a binary file is written, equivalent to `file.write(a.tobytes())`.

format : str

Format string for text file output. Each entry in the array is formatted to text by first converting it to the closest Python type, and then using "format" % item.

Notes

This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

tostring (*order='C'*)
Construct Python bytes containing the raw data bytes in the array.

Constructs Python bytes showing a copy of the raw contents of data memory. The bytes object can be produced in either ‘C’ or ‘Fortran’, or ‘Any’ order (the default is ‘C’-order). ‘Any’ order means C-order unless the `F_CONTIGUOUS` flag in the array is set, in which case it means ‘Fortran’ order.

This function is a compatibility alias for `tobytes`. Despite its name it returns bytes not strings.

Parameters

order : {‘C’, ‘F’, None}, optional

Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

Returns

s : bytes

Python bytes exhibiting a copy of a’s raw data.

Examples

```
>>> x = np.array([[0, 1], [2, 3]])
>>> x.tobytes()
b'\x00\x00\x00\x00\x01\x00\x00\x00\x02\x00\x00\x00\x03\x00\x00\x00'
>>> x.tobytes('C') == x.tobytes()
True
>>> x.tobytes('F')
b'\x00\x00\x00\x00\x02\x00\x00\x00\x01\x00\x00\x00\x03\x00\x00\x00'
```

trace (*offset=0, axis1=0, axis2=1, dtype=None, out=None*)

Return the sum along diagonals of the array.

Refer to `numpy.trace` for full documentation.

See also:

`numpy.trace`

equivalent function

var (*axis=None, dtype=None, out=None, ddof=0*)

Returns the variance of the array elements, along given axis.

Refer to `numpy.var` for full documentation.

See also:

`numpy.var`

equivalent function

Unit

class `astropy.units.Unit` (*st, represents=None, register=None, doc=None, format=None, namespace=None*)

Bases: `astropy.units.NamedUnit`

The main unit class.

There are a number of different ways to construct a `Unit`, but always returns a `UnitBase` instance. If the arguments refer to an already-existing unit, that existing unit instance is returned, rather than a new one.

- From a string:

```
Unit(s, format=None, parse_strict='silent')
```

Construct from a string representing a (possibly compound) unit.

The optional `format` keyword argument specifies the format the string is in, by default "generic". For a description of the available formats, see `astropy.units.format`.

The optional `parse_strict` keyword controls what happens when an unrecognized unit string is passed in. It may be one of the following:

- 'raise': (default) raise a `ValueError` exception.
- 'warn': emit a `Warning`, and return an `UnrecognizedUnit` instance.
- 'silent': return an `UnrecognizedUnit` instance.

•From a number:

```
Unit(number)
```

Creates a dimensionless unit.

•From a `UnitBase` instance:

```
Unit(unit)
```

Returns the given unit unchanged.

•From `None`:

```
Unit()
```

Returns the null unit.

•The last form, which creates a new `Unit` is described in detail below.

Parameters

st : str or list of str

The name of the unit. If a list, the first element is the canonical (short) name, and the rest of the elements are aliases.

represents : `UnitBase` instance

The unit that this named unit represents.

doc : str, optional

A docstring describing the unit.

format : dict, optional

A mapping to format-specific representations of this unit. For example, for the `Ohm` unit, it might be nice to have it displayed as `\Omega` by the `latex` formatter. In that case, `format` argument should be set to:

```
{ 'latex': r'\Omega' }
```

namespace : dictionary, optional

When provided, inject the unit (and all of its aliases) into the given namespace.

Raises

ValueError

If any of the given unit names are already in the registry.

ValueError

If any of the given unit names are not valid Python tokens.

Methods Summary

<code>decompose([bases])</code>	Return a unit object composed of only irreducible units.
<code>is_unity()</code>	Returns <code>True</code> if the unit is unscaled and dimensionless.

Methods Documentation

decompose (*bases*=`set([])`)

Return a unit object composed of only irreducible units.

Parameters

bases : sequence of `UnitBase`, optional

The bases to decompose into. When not provided, decomposes down to any irreducible units. When provided, the decomposed result will only contain the given units. This will raise a `UnitsError` if it's not possible to do so.

Returns

unit : `CompositeUnit` object

New object containing only irreducible unit objects.

is_unity ()

Returns `True` if the unit is unscaled and dimensionless.

UnitBase

class `astropy.units.UnitBase`

Bases: `object`

Abstract base class for units.

Most of the arithmetic operations on units are defined in this base class.

Should not be instantiated by users directly.

Attributes Summary

<code>aliases</code>	Returns the alias (long) names for this unit.
<code>bases</code>	Return the bases of the unit.
<code>cgs</code>	Returns a copy of the current <code>Unit</code> instance with CGS units.
<code>name</code>	Returns the canonical (short) name associated with this unit.
<code>names</code>	Returns all of the names associated with this unit.
<code>physical_type</code>	Return the physical type on the unit.
<code>powers</code>	Return the powers of the unit.
<code>scale</code>	Return the scale of the unit.
<code>si</code>	Returns a copy of the current <code>Unit</code> instance in SI units.

Methods Summary

<code>compose([equivalencies, units, max_depth, ...])</code>	Return the simplest possible composite unit(s) that represent the given unit.
<code>decompose([bases])</code>	Return a unit object composed of only irreducible units.
<code>find_equivalent_units([equivalencies, ...])</code>	Return a list of all the units that are the same type as <code>self</code> .
<code>get_converter(other[, equivalencies])</code>	Return the conversion function to convert values from <code>self</code> to the specified unit.
<code>in_units(other[, value, equivalencies])</code>	Alias for <code>to</code> for backward compatibility with <code>pynbody</code> .
<code>is_equivalent(other[, equivalencies])</code>	Returns <code>True</code> if this unit is equivalent to <code>other</code> .
<code>is_unity()</code>	Returns <code>True</code> if the unit is unscaled and dimensionless.
<code>to(other[, value, equivalencies])</code>	Return the converted values in the specified unit.
<code>to_string([format])</code>	Output the unit in the given format as a string.
<code>to_system(system)</code>	Converts this unit into ones belonging to the given system.

Attributes Documentation

aliases

Returns the alias (long) names for this unit.

bases

Return the bases of the unit.

cgs

Returns a copy of the current `Unit` instance with CGS units.

name

Returns the canonical (short) name associated with this unit.

names

Returns all of the names associated with this unit.

physical_type

Return the physical type on the unit.

Examples

```
>>> from astropy import units as u
>>> print(u.m.physical_type)
length
```

powers

Return the powers of the unit.

scale

Return the scale of the unit.

si

Returns a copy of the current `Unit` instance in SI units.

Methods Documentation

compose (*equivalencies*=[], *units*=None, *max_depth*=2, *include_prefix_units*=False)

Return the simplest possible composite unit(s) that represent the given unit. Since there may be multiple equally simple compositions of the unit, a list of units is always returned.

Parameters**equivalencies** : list of equivalence pairs, optional

A list of equivalence pairs to also list. See *Equivalencies*. This list is in addition to possible global defaults set by, e.g., `set_enabled_equivalencies`. Use `None` to turn off all equivalencies.

units : set of units to compose to, optional

If not provided, any known units may be used to compose into. Otherwise, `units` is a dict, module or sequence containing the units to compose into.

max_depth : int, optional

The maximum recursion depth to use when composing into composite units.

include_prefix_units : bool, optional

When `True`, include prefixed units in the result. Default is `False`.

Returns**units** : list of `CompositeUnit`

A list of candidate compositions. These will all be equally simple, but it may not be possible to automatically determine which of the candidates are better.

decompose (*bases=*`set([])`)

Return a unit object composed of only irreducible units.

Parameters**bases** : sequence of `UnitBase`, optional

The bases to decompose into. When not provided, decomposes down to any irreducible units. When provided, the decomposed result will only contain the given units. This will raise a `UnitsError` if it's not possible to do so.

Returns**unit** : `CompositeUnit` object

New object containing only irreducible unit objects.

find_equivalent_units (*equivalencies=*`[]`, *units=*`None`, *include_prefix_units=*`False`)

Return a list of all the units that are the same type as `self`.

Parameters**equivalencies** : list of equivalence pairs, optional

A list of equivalence pairs to also list. See *Equivalencies*. Any list given, including an empty one, supercedes global defaults that may be in effect (as set by `set_enabled_equivalencies`)

units : set of units to search in, optional

If not provided, all defined units will be searched for equivalencies. Otherwise, may be a dict, module or sequence containing the units to search for equivalencies.

include_prefix_units : bool, optional

When `True`, include prefixed units in the result. Default is `False`.

Returns**units** : list of `UnitBase`

A list of unit objects that match `u`. A subclass of `list` (`EquivalentUnitsList`) is returned that pretty-prints the list of units when output.

get_converter (*other*, *equivalencies*=[])

Return the conversion function to convert values from `self` to the specified unit.

Parameters

other : unit object or string

The unit to convert to.

equivalencies : list of equivalence pairs, optional

A list of equivalence pairs to try if the units are not directly convertible. See [Equivalencies](#). This list is in addition to possible global defaults set by, e.g., `set_enabled_equivalencies`. Use `None` to turn off all equivalencies.

Returns

func : callable

A callable that normally expects a single argument that is a scalar value or an array of values (or anything that may be converted to an array).

Raises

UnitsError

If units are inconsistent

in_units (*other*, *value*=1.0, *equivalencies*=[])

Alias for `to` for backward compatibility with `pynbody`.

is_equivalent (*other*, *equivalencies*=[])

Returns `True` if this unit is equivalent to `other`.

Parameters

other : unit object or string or tuple

The unit to convert to. If a tuple of units is specified, this method returns true if the unit matches any of those in the tuple.

equivalencies : list of equivalence pairs, optional

A list of equivalence pairs to try if the units are not directly convertible. See [Equivalencies](#). This list is in addition to possible global defaults set by, e.g., `set_enabled_equivalencies`. Use `None` to turn off all equivalencies.

Returns

bool

is_unity ()

Returns `True` if the unit is unscaled and dimensionless.

to (*other*, *value*=1.0, *equivalencies*=[])

Return the converted values in the specified unit.

Parameters

other : unit object or string

The unit to convert to.

value : scalar int or float, or sequence convertible to array, optional

Value(s) in the current unit to be converted to the specified unit. If not provided, defaults to 1.0

equivalencies : list of equivalence pairs, optional

A list of equivalence pairs to try if the units are not directly convertible. See *Equivalencies*. This list is in addition to possible global defaults set by, e.g., `set_enabled_equivalencies`. Use `None` to turn off all equivalencies.

Returns

values : scalar or array

Converted value(s). Input value sequences are returned as numpy arrays.

Raises**UnitsError**

If units are inconsistent

to_string (*format=u'generic'*)

Output the unit in the given format as a string.

Parameters

format : `astropy.units.format.Base` instance or str

The name of a format or a formatter object. If not provided, defaults to the generic format.

to_system (*system*)

Converts this unit into ones belonging to the given system. Since more than one result may be possible, a list is always returned.

Parameters

system : module

The module that defines the unit system. Commonly used ones include `astropy.units.si` and `astropy.units.cgs`.

To use your own module it must contain unit objects and a sequence member named `bases` containing the base units of the system.

Returns

units : list of `CompositeUnit`

The list is ranked so that units containing only the base units of that system will appear first.

UnitsError

exception `astropy.units.UnitsError`

The base class for unit-specific exceptions.

UnitsWarning

exception `astropy.units.UnitsWarning`

The base class for unit-specific exceptions.

UnrecognizedUnit

class `astropy.units.UnrecognizedUnit` (*st, register=None, doc=None, format=None, namespace=None*)

Bases: `astropy.units.IrreducibleUnit`

A unit that did not parse correctly. This allows for roundtripping it as a string, but no unit operations actually work on it.

Parameters

st : str

The name of the unit.

Methods Summary

<code>get_converter(other[, equivalencies])</code>	Return the conversion function to convert values from <code>self</code> to the specified unit.
<code>get_format_name(format)</code>	Get a name for this unit that is specific to a particular format.
<code>is_equivalent(other[, equivalencies])</code>	Returns <code>True</code> if this unit is equivalent to <code>other</code> .
<code>is_unity()</code>	Returns <code>True</code> if the unit is unscaled and dimensionless.
<code>to_string([format])</code>	Output the unit in the given format as a string.

Methods Documentation

get_converter (*other*, *equivalencies=None*)

Return the conversion function to convert values from `self` to the specified unit.

Parameters

other : unit object or string

The unit to convert to.

equivalencies : list of equivalence pairs, optional

A list of equivalence pairs to try if the units are not directly convertible. See [Equivalencies](#). This list is in addition to possible global defaults set by, e.g., `set_enabled_equivalencies`. Use `None` to turn off all equivalencies.

Returns

func : callable

A callable that normally expects a single argument that is a scalar value or an array of values (or anything that may be converted to an array).

Raises

UnitsError

If units are inconsistent

get_format_name (*format*)

Get a name for this unit that is specific to a particular format.

Uses the dictionary passed into the `format` kwarg in the constructor.

Parameters

format : str

The name of the format

Returns

name : str

The name of the unit for the given format.

is_equivalent (*other*, *equivalencies=None*)

Returns `True` if this unit is equivalent to `other`.

Parameters**other** : unit object or string or tuple

The unit to convert to. If a tuple of units is specified, this method returns true if the unit matches any of those in the tuple.

equivalencies : list of equivalence pairs, optional

A list of equivalence pairs to try if the units are not directly convertible. See *Equivalencies*. This list is in addition to possible global defaults set by, e.g., `set_enabled_equivalencies`. Use `None` to turn off all equivalencies.

Returns

bool

is_unity()Returns `True` if the unit is unscaled and dimensionless.**to_string** (*format=u'generic'*)

Output the unit in the given format as a string.

Parameters**format** : `astropy.units.format.Base` instance or str

The name of a format or a formatter object. If not provided, defaults to the generic format.

Class Inheritance Diagram**6.5.3 astropy.units.format Module**

A collection of different unit formats.

Functions

<code>get_format([format])</code>	Get a formatter by name.
-----------------------------------	--------------------------

get_format`astropy.units.format.get_format` (*format=None*)

Get a formatter by name.

Parameters**format** : str or `astropy.units.format.Base` instance or subclass

The name of the format, or the format instance or subclass itself.

Returns**format** : `astropy.units.format.Base` instance

The requested formatter.

Classes

<code>Base</code>	The abstract base class of all unit formats.
-------------------	--

Table 6.18 – continued from previous page

<code>Generic()</code>	A “generic” format.
<code>CDS()</code>	Support the Centre de Données astronomiques de Strasbourg Standards for Astronomical Catalogues 2.0 format, and
<code>Console()</code>	Output-only format for to display pretty formatting at the console.
<code>Fits()</code>	The FITS standard unit format.
<code>Latex()</code>	Output LaTeX to display the unit based on IAU style guidelines.
<code>OGIP()</code>	Support the units in Office of Guest Investigator Programs (OGIP) FITS files .
<code>Unicode()</code>	Output-only format to display pretty formatting at the console using Unicode characters.
<code>Unscaled()</code>	A format that doesn’t display the scale part of the unit, other than that, it is identical to the <code>Generic</code> format.
<code>VOUnit()</code>	The proposed IVOA standard for units used by the VO.

Base

class `astropy.units.format.Base`

Bases: `object`

The abstract base class of all unit formats.

Methods Summary

<code>parse(s)</code>	Convert a string to a unit object.
<code>to_string(u)</code>	Convert a unit object to a string.

Methods Documentation

parse (*s*)

Convert a string to a unit object.

to_string (*u*)

Convert a unit object to a string.

Generic

class `astropy.units.format.Generic`

Bases: `astropy.units.format.Base`

A “generic” format.

The syntax of the format is based directly on the FITS standard, but instead of only supporting the units that FITS knows about, it supports any unit available in the `astropy.units` namespace.

Methods Summary

<code>parse(s[, debug])</code>	Convert a string to a unit object.
<code>to_string(unit)</code>	Convert a unit object to a string.

Methods Documentation

parse (*s*, *debug=False*)
Convert a string to a unit object.

to_string (*unit*)
Convert a unit object to a string.

CDS

class `astropy.units.format.CDS`
Bases: `astropy.units.format.Base`

Support the Centre de Données astronomiques de Strasbourg Standards for Astronomical Catalogues 2.0 format, and the complete set of supported units. This format is used by VOTable up to version 1.2.

Methods Summary

<code>parse(s[, debug])</code>	Convert a string to a unit object.
<code>to_string(unit)</code>	Convert a unit object to a string.

Methods Documentation

parse (*s*, *debug=False*)
Convert a string to a unit object.

to_string (*unit*)
Convert a unit object to a string.

Console

class `astropy.units.format.Console`
Bases: `astropy.units.format.Base`

Output-only format for to display pretty formatting at the console.

For example:

```
>>> import astropy.units as u
>>> print(u.Ry.decompose().to_string('console'))
          m^2 kg
2.1798721*10^-18 -----
                   s^2
```

Methods Summary

<code>to_string(unit)</code>	Convert a unit object to a string.
------------------------------	------------------------------------

Methods Documentation

to_string (*unit*)
Convert a unit object to a string.

Fits

class `astropy.units.format.Fits`
Bases: `astropy.units.format.Generic`

The FITS standard unit format.

This supports the format defined in the Units section of the [FITS Standard](#).

Attributes Summary

<code>name</code>	<code>unicode(string [, encoding[, errors]]) -> object</code>
-------------------	--

Methods Summary

<code>to_string(unit)</code>	Convert a unit object to a string.
------------------------------	------------------------------------

Attributes Documentation

`name = u'fits'`

Methods Documentation

to_string (*unit*)
Convert a unit object to a string.

Latex

class `astropy.units.format.Latex`
Bases: `astropy.units.format.Base`

Output LaTeX to display the unit based on IAU style guidelines.

Attempts to follow the [IAU Style Manual](#).

Methods Summary

<code>to_string(unit)</code>	Convert a unit object to a string.
------------------------------	------------------------------------

Methods Documentation

to_string (*unit*)
Convert a unit object to a string.

OGIP

class `astropy.units.format.OGIP`
Bases: `astropy.units.format.Generic`
Support the units in Office of Guest Investigator Programs (OGIP) FITS files.

Methods Summary

<code>parse(s[, debug])</code>	Convert a string to a unit object.
<code>to_string(unit)</code>	Convert a unit object to a string.

Methods Documentation

parse (*s*, *debug=False*)
Convert a string to a unit object.

to_string (*unit*)
Convert a unit object to a string.

Unicode

class `astropy.units.format.Unicode`
Bases: `astropy.units.format.Console`
Output-only format to display pretty formatting at the console using Unicode characters.

For example:

```
>>> import astropy.units as u
>>> print(u.Ry.decompose().to_string('unicode'))
2.1798721×10-18  $\frac{\text{m}^2 \text{ kg}}{\text{s}^2}$ 
```

Unscaled

class `astropy.units.format.Unscaled`
Bases: `astropy.units.format.Generic`
A format that doesn't display the scale part of the unit, other than that, it is identical to the `Generic` format.
This is used in some error messages where the scale is irrelevant.

VOUnit

class `astropy.units.format.VOUnit`

Bases: `astropy.units.format.Generic`

The proposed IVOA standard for units used by the VO.

This is an implementation of [proposed IVOA standard for units](#).

Methods Summary

<code>parse(s[, debug])</code>	Convert a string to a unit object.
<code>to_string(unit)</code>	Convert a unit object to a string.

Methods Documentation

parse (*s*, *debug=False*)

Convert a string to a unit object.

to_string (*unit*)

Convert a unit object to a string.

Class Inheritance Diagram

6.5.4 astropy.units.si Module

This package defines the SI units. They are also available in the `astropy.units` namespace.

Table 6.28: Available Units

Unit	Description	Represents	Alias
A	ampere: base unit of electric current in SI		amp
a	annum (a)	365.25 d	ann
Angstrom	ångström: 10^{-10} m	0.1 nm	Å, ÅA,
arcmin	arc minute: angular measurement	0.016666667°	arc
arcsec	arc second: angular measurement	0.00027777778°	arc
bar	bar: pressure	100000 Pa	
Bq	becquerel: unit of radioactivity	Hz	bec
C	coulomb: electric charge	A s	cou
cd	candela: base unit of luminous intensity in SI		can
Ci	curie: unit of radioactivity	$2.7027027 \times 10^{-11}$ Bq	cur
d	day (d)	24 h	day
deg	degree: angular measurement 1/360 of full rotation	0.017453293 rad	deg
deg_C	Degrees Celsius		Cel
eV	Electron Volt	$1.6021766 \times 10^{-19}$ J	ele
F	Farad: electrical capacitance	$\frac{C}{V}$	Far
fortnight	fortnight	2 wk	
g	gram (g)	0.001 kg	gra
H	Henry: inductance	$\frac{Wb}{A}$	Her
h	hour (h)	3600 s	hou
hourangle	hour angle: angular measurement with 24 in a full circle	15°	

Table 6.28 – continued from previous page

Unit	Description	Represents	Alias
Hz	Frequency	$\frac{1}{s}$	Herz
J	Joule: energy	N m	Joule
K	Kelvin: temperature with a null point at absolute zero.		Kelvin
kg	kilogram: base unit of mass in SI.		kilogram
l	liter: metric unit of volume	1000 cm ³	L, l
lm	lumen: luminous flux	cd sr	lumen
lx	lux: luminous emittance	$\frac{lm}{m^2}$	lux
m	meter: base unit of length in SI		metre
mas	arc second: angular measurement	0.001 ''	
micron	micron: alias for micrometer (um)	μm	
min	minute (min)	60 s	minute
mol	mole: amount of a chemical substance in SI.		mole
N	Newton: force	$\frac{kg\ m}{s^2}$	Newton
Ohm	Ohm: electrical resistance	$\frac{V}{A}$	ohm
Pa	Pascal: pressure	$\frac{J}{m^3}$	Pascal
%	percent: one hundredth of unity, factor 0.01	0.01	percent
rad	radian: angular measurement of the ratio between the length on an arc and its radius		radian
S	Siemens: electrical conductance	$\frac{A}{V}$	Siemens
s	second: base unit of time in SI.		second
sday	Sidereal day (sday) is the time of one rotation of the Earth.	86164.091 s	
sr	steradian: unit of solid angle in SI	rad ²	steradian
t	Metric tonne	1000 kg	tonne
T	Tesla: magnetic flux density	$\frac{Wb}{m^2}$	Tesla
uas	arc second: angular measurement	$1 \times 10^{-6} ''$	
V	Volt: electric potential or electromotive force	$\frac{J}{C}$	Volt
W	Watt: power	$\frac{J}{s}$	Watt
Wb	Weber: magnetic flux	V s	Weber
wk	week (wk)	7 d	week
yr	year (yr)	365.25 d	year

6.5.5 astropy.units.cgs Module

This package defines the CGS units. They are also available in the top-level `astropy.units` namespace.

Table 6.29: Available Units

Unit	Description	Represents	Aliases	SI Pre-fixes
abC	abcoulomb: CGS (EMU) of charge	Bi s	abcoulomb	N
Ba	Barye: CGS unit of pressure	$\frac{\text{g}}{\text{cm s}^2}$	Barye, barye	N
Bi	Biot: CGS (EMU) unit of current	$\frac{\text{cm}^{1/2} \text{g}^{1/2}}{\text{s}}$	Biot, abA, abampere, emu	N
C	coulomb: electric charge	A s	coulomb	N
cd	candela: base unit of luminous intensity in SI		candela	N
cm	centimeter (cm)	cm	centimeter	N
D	Debye: CGS unit of electric dipole moment	$3.3333333 \times 10^{-30} \text{C m}$	Debye, debye	N
deg_C	Degrees Celsius		Celsius	N
dyn	dyne: CGS unit of force	$\frac{\text{cm g}}{\text{s}^2}$	dyne	N
erg	erg: CGS unit of energy	$\frac{\text{cm}^2 \text{g}}{\text{s}^2}$		N
Fr	Franklin: CGS (ESU) unit of charge	$\frac{\text{cm}^{3/2} \text{g}^{1/2}}{\text{s}}$	Franklin, statcoulomb, statC, esu	N
G	Gauss: CGS unit for magnetic field	0.0001 T	Gauss, gauss	N
g	gram (g)	0.001 kg	gram	N
Gal	Gal: CGS unit of acceleration	$\frac{\text{cm}}{\text{s}^2}$	gal	N
K	Kelvin: temperature with a null point at absolute zero.		Kelvin	N
k	kayser: CGS unit of wavenumber	$\frac{1}{\text{cm}}$	Kayser, kayser	N
mol	mole: amount of a chemical substance in SI.		mole	N
P	poise: CGS unit of dynamic viscosity	$\frac{\text{g}}{\text{cm s}}$	poise	N
rad	radian: angular measurement of the ratio between the length on an arc and its radius		radian	N
s	second: base unit of time in SI.		second	N
sr	steradian: unit of solid angle in SI	rad ²	steradian	N
St	stokes: CGS unit of kinematic viscosity	$\frac{\text{cm}^2}{\text{s}}$	stokes	N
statA	statampere: CGS (ESU) unit of current	$\frac{\text{Fr}}{\text{s}}$	statampere	N

6.5.6 astropy.units.astrophys Module

This package defines the astrophysics-specific units. They are also available in the `astropy.units` namespace.

The `mag` unit is provided for compatibility with the FITS unit string standard. However, it is not very useful as-is since it is “orphaned” and can not be converted to any other unit. A future astropy magnitudes library is planned to address this shortcoming.

Table 6.30: Available Units

Unit	Description	Represents	Aliases	SI Pre- fixes
adu	adu			N
AU	astronomical unit: approximately the mean Earth–Sun distance.	1.4959787×10^{11}	mau	N
barn	barn: unit of area used in HEP	$1 \times 10^{-28} \text{ m}^2$		N
beam	beam			N
bin	bin			N
bit	b (bit)		b, bit	Y
byte	B (byte)		B, byte	Y
chan	chan			N
ct	count (ct)		count	N
cycle	cycle: angular measurement, a full turn or rotation	6.2831853 rad	cy	N
dB	Decibel: ten per base 10 logarithmic unit	0.1 dex	decibel	N
dex	Dex: Base 10 logarithmic unit			Y
electron	Number of electrons			N
Jy	Jansky: spectral flux density	$1 \times 10^{-26} \frac{\text{W}}{\text{Hz m}^2}$	Jansky, jansky	N
lyr	Light year	9.4607305×10^{15}	lightyear	N
M_e	Electron mass	$9.1093829 \times 10^{-31} \text{ kg}$		N
M_p	Proton mass	$1.6726218 \times 10^{-27} \text{ kg}$		N
mag	Astronomical magnitude: -2.5 per base 10 logarithmic unit	-0.4 dex		N
pc	parsec: approximately 3.26 light-years.	3.0856776×10^{16}	parsec	N
ph	photon (ph)		photon	Y
pix	pixel (pix)		pixel	N
R	Rayleigh: photon flux	7.9577472×10^8	ph Rayleigh, rayleigh	N
Ry	Rydberg: Energy of a photon whose wavenumber is the Rydberg constant	13.605692 eV	rydberg	N
solLum	Solar luminance	$3.846 \times 10^{26} \text{ W}$	L_sun, Lsun	N
solMass	Solar mass	$1.9891 \times 10^{30} \text{ kg}$	M_sun, Msun	N
solRad	Solar radius	$6.95508 \times 10^8 \text{ m}$	R_sun, Rsun	N
Sun	Sun			N
u	Unified atomic mass unit	$1.6605387 \times 10^{-27} \text{ kg}$	u, Dalton	N
vox	voxel (vox)		voxel	N

6.5.7 astropy.units.imperial Module

This package defines colloquially used Imperial units. By default, they are not enabled. To enable them, do:

```
>>> from astropy.units import imperial
>>> imperial.enable()
```

Table 6.31: Available Units

Unit	Description	Repre- sents	Aliases	SI Pre- fixes
ac	International acre	43560 ft ²	acre	N
BTU	British thermal unit	1.0550559 kJ	btu	N
cal	Thermochemical calorie: pre-SI metric unit of energy	4.184 J	calorie	N
cup	U.S.	0.5 pint		N
deg_F	Degrees Fahrenheit		Fahrenheit	N
foz	U.S.	0.125 cup	fluid_oz, fluid_ounce	N
ft	International foot	12 inch	foot	N
gallon	U.S.	3.7854118 ↓		N
hp	Electrical horsepower	745.69987 W	horsepower	N
inch	International inch	2.54 cm		N
kcal	Calorie: colloquial definition of Calorie	1000 cal	Cal, Calorie, kilocal, kilocalorie	N
kn	nautical unit of speed: 1 nmi per hour	$\frac{\text{nmi}}{\text{h}}$	kt, knot, NMPH	N
lb	International avoirdupois pound	16 oz	pound	N
mi	International mile	5280 ft	mile	N
nmi	Nautical mile	1852 m	nauticalmile, NM	N
oz	International avoirdupois ounce	28.349523 g	ounce	N
pint	U.S.	0.5 quart		N
quart	U.S.	0.25 gallon		N
tbsp	U.S.	0.5 foz	tablespoon	N
ton	International avoirdupois ton	2000 lb		N
tsp	U.S.	0.33333333 tsp	teaspoon	N
yd	International yard	3 ft	yard	N

Functions

`enable()` Enable Imperial units so they appear in results of `find_equivalent_units` and `compose`.

enable

```
astropy.units.imperial.enable()
```

Enable Imperial units so they appear in results of `find_equivalent_units` and `compose`.

This may be used with the `with` statement to enable Imperial units only temporarily.

6.5.8 astropy.units.cds Module

This package defines units used in the CDS format.

Contains the units defined in [Centre de Données astronomiques de Strasbourg Standards for Astronomical Catalogues 2.0 format](#), and the complete set of supported units. This format is used by `VOTable` up to version 1.2.

To include them in `compose` and the results of `find_equivalent_units`, do:

```
>>> from astropy.units import cds
>>> cds.enable()
```

Table 6.33: Available Units

Unit	Description	Represents	Aliases	SI Prefixes
%	percent	%		N
---	dimensionless and unscaled			N
\h	Planck constant	$6.6260696 \times 10^{-34} \text{ J s}$		N
A	Ampere	A		Y
a	year	a		N
a0	Bohr radius	$5.2917721 \times 10^{-11} \text{ m}$		N
AA	Angstrom	$\overset{\circ}{\text{A}}$	Å, Angstrom, Angstroem	N
al	Light year	lyr		N
alpha	Fine structure constant	0.0072973526		N
arcmin	minute of arc	'	arcmin	N
arcsec	second of arc	"	arcsec	N
atm	atmosphere	101325 Pa		N
AU	astronomical unit	AU	au	N
bar	bar	bar		N
barn	barn	barn		N
bit	bit	bit		Y
byte	byte	byte		Y
C	Coulomb	C		N
c	speed of light	$2.9979246 \times 10^8 \frac{\text{m}}{\text{s}}$		N
cal	calorie	4.1854 J		N
cd	candela	cd		Y
Crab	Crab (X-ray) flux			Y
ct	count	ct		Y
D	Debye (dipole)	D		N
d	Julian day	d		N
deg	degree	°	°, degree	N
dyn	dyne	dyn		N
e	electron charge	$1.6021766 \times 10^{-19} \text{ C}$		N
eps0	electric constant	$8.8541878 \times 10^{-12} \frac{\text{F}}{\text{m}}$		N
erg	erg	erg		N
eV	electron volt	eV		N
F	Farad	F		N
G	Gravitation constant	$6.67384 \times 10^{-11} \frac{\text{m}^3}{\text{kg s}^2}$		N
g	gram	g		N
gauss	Gauss	G		N
geoMass	Earth mass	$5.9742 \times 10^{24} \text{ kg}$	Mgeo	N
H	Henry	H		N
h	hour	h		N
hr	hour	h		N
Hz	Hertz	Hz		N
inch	inch	0.0254 m		N
J	Joule	J		N
JD	Julian day	d		N
jovMass	Jupiter mass	$1.8987 \times 10^{27} \text{ kg}$	Mjup	N
Jy	Jansky	Jy		N
k	Boltzmann	$1.3806488 \times 10^{-23} \frac{\text{J}}{\text{K}}$		N
K	Kelvin	K		Y
l	litre	↑		N
lm	lumen	lm		N

Continued on next page

Table 6.33 – continued from previous page

Unit	Description	Represents	Aliases	SI Prefixes
Lsun	solar luminosity	L_{\odot}	solLum	N
lx	lux	lx		N
lyr	Light year	lyr		N
m	meter	m		Y
mag	magnitude	mag		N
mas	millisecond of arc	marcsec		N
me	electron mass	$9.1093829 \times 10^{-31}$ kg		N
min	minute	min		N
MJD	Julian day	d		N
mmHg	millimeter of mercury	133.32239 Pa		N
mol	mole	mol		Y
mp	proton mass	$1.6726218 \times 10^{-27}$ kg		N
Msun	solar mass	M_{\odot}	solMass	N
mu0	magnetic constant	$1.2566371 \times 10^{-6} \frac{N}{A^2}$	μ_0	N
muB	Bohr magneton	$9.2740097 \times 10^{-24} \frac{J}{T}$		N
N	Newton	N		N
Ohm	Ohm	Ω		N
Pa	Pascal	Pa		N
pc	parsec	pc		N
ph	photon	ph		Y
pi	π	3.1415927		N
pix	pixel	pix		Y
ppm	parts per million	1×10^{-6}		N
R	gas constant	$8.3144621 \frac{J}{K mol}$		N
rad	radian	rad		Y
Rgeo	Earth equatorial radius	6378136 m		N
Rjup	Jupiter equatorial radius	71492000 m		N
Rsun	solar radius	R_{\odot}	solRad	N
Ry	Rydberg	R_{∞}		N
s	second	s	sec	Y
S	Siemens	S		N
sr	steradian	sr		N
Sun	solar unit	Sun		Y
T	Tesla	T		N
t	metric tonne	1000 kg		N
u	atomic mass	$1.6605389 \times 10^{-27}$ kg		N
V	Volt	V		N
W	Watt	W		N
Wb	Weber	Wb		N
yr	year	a		N
uas	microsecond of arc	$\mu arcsec$		N

Functions

`enable()` Enable CDS units so they appear in results of `find_equivalent_units` and `compose`.

enable

`astropy.units.cds.enable()`

Enable CDS units so they appear in results of `find_equivalent_units` and `compose`.

This may be used with the `with` statement to enable CDS units only temporarily.

6.5.9 astropy.units.equivalencies Module

A set of standard astronomical equivalencies.

Functions

<code>parallax()</code>	Returns a list of equivalence pairs that handle the conversion between parallax and distance.
<code>spectral()</code>	Returns a list of equivalence pairs that handle spectral wavelength, wave number, frequency, and energy equivalences.
<code>spectral_density(wav[, factor])</code>	Returns a list of equivalence pairs that handle spectral density with regard to wavelength and frequency.
<code>doppler_radio(rest)</code>	Return the equivalency pairs for the radio convention for velocity.
<code>doppler_optical(rest)</code>	Return the equivalency pairs for the optical convention for velocity.
<code>doppler_relativistic(rest)</code>	Return the equivalency pairs for the relativistic convention for velocity.
<code>mass_energy()</code>	Returns a list of equivalence pairs that handle the conversion between mass and energy.
<code>brightness_temperature(beam_area, disp)</code>	Defines the conversion between Jy/beam and “brightness temperature”, T_B , in K.
<code>dimensionless_angles()</code>	Allow angles to be equivalent to dimensionless (with $1 \text{ rad} = 1 \text{ m/m} = 1$).
<code>logarithmic()</code>	Allow logarithmic units to be converted to dimensionless fractions.
<code>temperature()</code>	Convert between Kelvin, Celsius, and Fahrenheit here because Unit and Compose don't handle them.
<code>temperature_energy()</code>	Convert between Kelvin and keV(eV) to an equivalent amount.

parallax

`astropy.units.equivalencies.parallax()`

Returns a list of equivalence pairs that handle the conversion between parallax angle and distance.

spectral

`astropy.units.equivalencies.spectral()`

Returns a list of equivalence pairs that handle spectral wavelength, wave number, frequency, and energy equivalences.

Allows conversions between wavelength units, wave number units, frequency units, and energy units as they relate to light.

There are two types of wave number:

- spectroscopic - $1/\lambda$ (per meter)
- angular - $2\pi/\lambda$ (radian per meter)

spectral_density

`astropy.units.equivalencies.spectral_density(wav, factor=None)`

Returns a list of equivalence pairs that handle spectral density with regard to wavelength and frequency.

Parameters**wav**: `Quantity``Quantity` associated with values being converted (e.g., wavelength or frequency).**Notes**

The `factor` argument is left for backward-compatibility with the syntax `spectral_density(unit, factor)` but users are encouraged to use `spectral_density(factor * unit)` instead.

doppler_radio`astropy.units.equivalencies.doppler_radio` (*rest*)

Return the equivalency pairs for the radio convention for velocity.

The radio convention for the relation between velocity and frequency is:

$$V = c \frac{f_0 - f}{f_0}; f(V) = f_0(1 - V/c)$$

Parameters**rest**: `Quantity`

Any quantity supported by the standard spectral equivalencies (wavelength, energy, frequency, wave number).

References[NRAO site defining the conventions](#)**Examples**

```
>>> import astropy.units as u
>>> CO_restfreq = 115.27120*u.GHz # rest frequency of 12 CO 1-0 in GHz
>>> radio_CO_equiv = u.doppler_radio(CO_restfreq)
>>> measured_freq = 115.2832*u.GHz
>>> radio_velocity = measured_freq.to(u.km/u.s, equivalencies=radio_CO_equiv)
>>> radio_velocity
<Quantity -31.209092088877583 km / s>
```

doppler_optical`astropy.units.equivalencies.doppler_optical` (*rest*)

Return the equivalency pairs for the optical convention for velocity.

The optical convention for the relation between velocity and frequency is:

$$V = c \frac{f_0 - f}{f}; f(V) = f_0(1 + V/c)^{-1}$$

Parameters**rest**: `Quantity`

Any quantity supported by the standard spectral equivalencies (wavelength, energy, frequency, wave number).

References

NRAO site defining the conventions

Examples

```
>>> import astropy.units as u
>>> CO_restfreq = 115.27120*u.GHz # rest frequency of 12 CO 1-0 in GHz
>>> optical_CO_equiv = u.doppler_optical(CO_restfreq)
>>> measured_freq = 115.2832*u.GHz
>>> optical_velocity = measured_freq.to(u.km/u.s, equivalencies=optical_CO_equiv)
>>> optical_velocity
<Quantity -31.20584348799674 km / s>
```

doppler_relativistic

`astropy.units.equivalencies.doppler_relativistic` (*rest*)

Return the equivalency pairs for the relativistic convention for velocity.

The full relativistic convention for the relation between velocity and frequency is:

$$V = c \frac{f_0^2 - f^2}{f_0^2 + f^2}; f(V) = f_0 \frac{(1 - (V/c)^2)^{1/2}}{(1 + V/c)}$$

Parameters

rest: `Quantity`

Any quantity supported by the standard spectral equivalencies (wavelength, energy, frequency, wave number).

References

NRAO site defining the conventions

Examples

```
>>> import astropy.units as u
>>> CO_restfreq = 115.27120*u.GHz # rest frequency of 12 CO 1-0 in GHz
>>> relativistic_CO_equiv = u.doppler_relativistic(CO_restfreq)
>>> measured_freq = 115.2832*u.GHz
>>> relativistic_velocity = measured_freq.to(u.km/u.s, equivalencies=relativistic_CO_equiv)
>>> relativistic_velocity
<Quantity -31.207467619351537 km / s>
>>> measured_velocity = 1250 * u.km/u.s
>>> relativistic_frequency = measured_velocity.to(u.GHz, equivalencies=relativistic_CO_equiv)
>>> relativistic_frequency
<Quantity 114.79156866993588 GHz>
>>> relativistic_wavelength = measured_velocity.to(u.mm, equivalencies=relativistic_CO_equiv)
>>> relativistic_wavelength
<Quantity 2.6116243681798923 mm>
```

mass_energy

`astropy.units.equivalencies.mass_energy()`

Returns a list of equivalence pairs that handle the conversion between mass and energy.

brightness_temperature

`astropy.units.equivalencies.brightness_temperature(beam_area, disp)`

Defines the conversion between Jy/beam and “brightness temperature”, T_B , in Kelvins. The brightness temperature is a unit very commonly used in radio astronomy. See, e.g., “Tools of Radio Astronomy” (Wilson 2009) eqn 8.16 and eqn 8.19 (these pages are available on [google books](#)).

$$T_B \equiv S_\nu / (2k\nu^2/c^2)$$

However, the beam area is essential for this computation: the brightness temperature is inversely proportional to the beam area

Parameters

beam_area : Beam Area equivalent

Beam area in angular units, i.e. steradian equivalent

disp : `Quantity` with spectral units

The observed `spectral` equivalent `Unit` (e.g., frequency or wavelength)

Examples

Arecibo C-band beam:

```
>>> import numpy as np
>>> from astropy import units as u
>>> beam_area = np.pi*(50*u.arcsec)**2
>>> freq = 5*u.GHz
>>> equiv = u.brightness_temperature(beam_area, freq)
>>> u.Jy.to(u.K, equivalencies=equiv)
7.052588858846446
>>> (1*u.Jy).to(u.K, equivalencies=equiv)
<Quantity 7.052588858846446 K>
```

VLA synthetic beam:

```
>>> beam_area = np.pi*(15*u.arcsec)**2
>>> freq = 5*u.GHz
>>> equiv = u.brightness_temperature(beam_area, freq)
>>> u.Jy.to(u.K, equivalencies=equiv)
78.36209843162719
```

dimensionless_angles

`astropy.units.equivalencies.dimensionless_angles()`

Allow angles to be equivalent to dimensionless (with 1 rad = 1 m/m = 1).

It is special compared to other equivalency pairs in that it allows this independent of the power to which the angle is raised, and independent of whether it is part of a more complicated unit.

logarithmic

```
astropy.units.equivalencies.logarithmic()
```

Allow logarithmic units to be converted to dimensionless fractions

temperature

```
astropy.units.equivalencies.temperature()
```

Convert between Kelvin, Celsius, and Fahrenheit here because Unit and CompositeUnit cannot do addition or subtraction properly.

temperature_energy

```
astropy.units.equivalencies.temperature_energy()
```

Convert between Kelvin and keV(eV) to an equivalent amount.

6.6 Acknowledgments

This code is adapted from the [pynbody](#) units module written by Andrew Pontzen, who has granted the Astropy project permission to use the code under a BSD license.

N-DIMENSIONAL DATASETS (ASTROPY.NDDATA)

7.1 Introduction

`astropy.nddata` provides the `NDData` class and related tools to manage n-dimensional array-based data (e.g. CCD images, IFU data, grid-based simulation data, ...). This is more than just `numpy.ndarray` objects, because it provides metadata that cannot be easily provided by a single array.

Note: The `NDData` class is still under development, and support for WCS and units is not yet implemented.

7.2 Getting started

An `NDData` object can be instantiated by passing it an n-dimensional Numpy array:

```
>>> import numpy as np
>>> from astropy.nddata import NDData
>>> array = np.zeros((12, 12, 12)) # a 3-dimensional array with all zeros
>>> ndd = NDData(array)
```

This object has a few attributes in common with Numpy:

```
>>> ndd.ndim
3
>>> ndd.shape
(12, 12, 12)
>>> ndd.dtype
dtype('float64')
```

The underlying Numpy array can be accessed via the `data` attribute:

```
>>> ndd.data
array([[[ 0.,  0.,  0., ...
...
...]])
```

Values can be masked using the `mask` attribute, which should be a boolean Numpy array with the same dimensions as the data, e.g.:

```
>>> ndd.mask = ndd.data > 0.9
```

A mask value of `True` indicates a value that should be ignored, while a mask value of `False` indicates a valid value.

Similarly, attributes are available to store generic meta-data, flags, and uncertainties, and the `NDData` class includes methods to combine datasets with arithmetic operations (which include uncertainties propagation). These are described in *NDData overview*.

7.3 Using `nddata`

7.3.1 NDData overview

Initializing

An `NDData` object can be instantiated by passing it an n-dimensional Numpy array:

```
>>> import numpy as np
>>> from astropy.nddata import NDData
>>> array = np.zeros((12, 12, 12)) # a 3-dimensional array with all zeros
>>> ndd = NDData(array)
```

Note that the data in `ndd` is a reference to the original `array`, so changing the data in `ndd` will change the corresponding data in `array` in most circumstances.

An `NDData` object can also be instantiated by passing it an `NDData` object:

```
>>> ndd1 = NDData(array)
>>> ndd2 = NDData(ndd1)
```

As above, the data in “`ndd2`” is a reference to the data in `ndd1`, so changes to one will affect the other.

This object has a few attributes in common with Numpy:

```
>>> ndd.ndim
3
>>> ndd.shape
(12, 12, 12)
>>> ndd.dtype
dtype('float64')
```

The underlying Numpy array can be accessed via the `data` attribute:

```
>>> ndd.data
array([[[[ 0.,  0.,  0., ...
```

Mask

Values can be masked using the `mask` attribute, which should be a boolean Numpy array with the same dimensions as the data, e.g.:

```
>>> ndd.mask = ndd.data > 0.9
```

A mask value of `True` indicates a value that should be ignored, while a mask value of `False` indicates a valid value.

Flags

Values can be assigned one or more flags. The `flags` attribute is used to store either a single Numpy array (of any type) with dimensions matching that of the data, or a `FlagCollection`, which is essentially a dictionary of Numpy arrays (of any type) with the same shape as the data. The following example demonstrates setting a single set of integer flags:

```
>>> ndd.flags = np.zeros(ndd.shape)
>>> ndd.flags[ndd.data < 0.1] = 1
>>> ndd.flags[ndd.data < 0.01] = 2
```

but one can also have multiple flag layers with different types:

```
>>> from astropy.nddata import FlagCollection
>>> ndd.flags = FlagCollection(shape=(12, 12, 12))
>>> ndd.flags['photometry'] = np.zeros(ndd.shape, dtype=str)
>>> ndd.flags['photometry'][ndd.data > 0.9] = 's'
>>> ndd.flags['cosmic_rays'] = np.zeros(ndd.shape, dtype=int)
>>> ndd.flags['cosmic_rays'][ndd.data > 0.99] = 99
```

and flags can easily be used to set the mask:

```
>>> ndd.mask = ndd.flags['cosmic_rays'] == 99
```

Uncertainties

`NDData` objects have an `uncertainty` attribute that can be used to set the uncertainty on the data values. This is done by using classes to represent the uncertainties of a given type. For example, to set standard deviation uncertainties on the pixel values, you can do:

```
>>> from astropy.nddata import StdDevUncertainty
>>> ndd.uncertainty = StdDevUncertainty(np.ones((12, 12, 12)) * 0.1)
```

Note: For information on creating your own uncertainty classes, see *Subclassing `NDData` and `NDUncertainty`*.

Arithmetic

Provided that the world coordinate system (WCS) and shape match, and that the units are consistent, two `NDData` instances can be added, subtracted, multiplied or divided from each other, with uncertainty propagation, creating a new `NDData` object:

```
ndd3 = ndd1.add(ndd2)
ndd4 = ndd1.subtract(ndd2)
ndd5 = ndd1.multiply(ndd2)
ndd6 = ndd1.divide(ndd2)
```

The purpose of the `add()`, `subtract()`, `multiply()` and `divide()` methods is to allow the combination of two data objects that have common WCS and shape and units consistent with the operation performed, with consistent behavior for masks, and with a framework to propagate uncertainties. Currently any flags on the operands are dropped so that the result of the operation always has no flags. These methods are intended for use by sub-classes and functions that deal with more complex combinations.

Entries that are masked in either of the operands are also masked in the result.

Warning: Uncertainty propagation is still experimental, and does not take into account correlated uncertainties.

Meta-data

The `NDData` class includes a `meta` attribute that defaults to an empty dictionary, and can be used to set overall meta-data for the dataset:

```
ndd.meta['exposure_time'] = 340.
ndd.meta['filter'] = 'J'
```

Elements of the meta-data dictionary can be set to any valid Python object:

```
ndd.meta['history'] = ['calibrated', 'aligned', 'flat-fielded']
```

Converting to Numpy arrays

`NDData` objects can also be easily converted to numpy arrays:

```
>>> import numpy as np
>>> arr = np.array(ndd)
>>> np.all(arr == mydataarray)
True
```

If a mask is defined, this will result in a `MaskedArray`, so in all cases a useable `numpy.ndarray` or subclass will result. This allows straightforward plotting of `NDData` objects with 1- and 2-dimensional datasets using Matplotlib:

```
>>> from matplotlib import pyplot as plt
>>> plt.plot(ndd)
```

This works because the Matplotlib plotting functions automatically convert their inputs using `numpy.array`.

7.3.2 Subclassing `NDData` and `NDUncertainty`

Subclassing `NDUncertainty`

New error classes should sub-class from `NDUncertainty`, and should provide methods with the following API:

```
class MyUncertainty(NDUncertainty):

    def propagate_add(self, other_nddata, result_data):
        ...
        result_uncertainty = MyUncertainty(...)
        return result_uncertainty

    def propagate_subtract(self, other_nddata, result_data):
        ...
        result_uncertainty = MyUncertainty(...)
        return result_uncertainty

    def propagate_multiply(self, other_nddata, result_data):
        ...
        result_uncertainty = MyUncertainty(...)
        return result_uncertainty

    def propagate_divide(self, other_nddata, result_data):
        ...
        result_uncertainty = MyUncertainty(...)
        return result_uncertainty
```

All error sub-classes inherit an attribute `self.parent_nddata` that is automatically set to the parent `NDData` object that they are attached to. The arguments passed to the error propagation methods are `other_nddata`, which is the `NDData` object that is being combined with `self.parent_nddata`, and `result_data`, which is a Numpy array that contains the data array after the arithmetic operation. All these methods should return an error instance `result_uncertainty`, and should not modify `parent_nddata` directly. For subtraction and division, the order of the operations is `parent_nddata - other_nddata` and `parent_nddata / other_nddata`.

To make it easier and clearer to code up the error propagation, you can use variables with more explicit names, e.g:

```

class MyUncertainty (NDUncertainty):

    def propogate_add(self, other_nddata, result_data):

        left_uncertainty = self.parent.uncertainty.array
        right_uncertainty = other_nddata.uncertainty.array

        ...

```

Note that the above example assumes that the errors are stored in an `array` attribute, but this does not have to be the case.

For an example of a complete implementation, see `StdDevUncertainty`.

7.4 Reference/API

7.4.1 astropy.nddata Module

The `astropy.nddata` subpackage provides the `NDData` class and related tools to manage n-dimensional array-based data (e.g. CCD images, IFU Data, grid-based simulation data, ...). This is more than just `numpy.ndarray` objects, because it provides metadata that cannot be easily provided by a single array.

Classes

<code>Conf</code>	Configuration parameters for <code>astropy.nddata</code> .
<code>FlagCollection(*args, **kwargs)</code>	The purpose of this class is to provide a dictionary for containing arrays of flags
<code>IncompatibleUncertaintiesException</code>	This exception should be used to indicate cases in which uncertainties with two
<code>MissingDataAssociationException</code>	This exception should be used to indicate that an uncertainty instance has not be
<code>NDData(data[, uncertainty, mask, flags, ...])</code>	A Superclass for array-based data in Astropy.
<code>NDUncertainty</code>	This is the base class for uncertainty classes used with <code>NDData</code> .
<code>StdDevUncertainty([array, copy])</code>	A class for standard deviation uncertainties

Conf

class `astropy.nddata.Conf`
 Bases: `astropy.config.ConfigNamespace`
 Configuration parameters for `astropy.nddata`.

Attributes Summary

<code>warn_setting_unit_directly</code>	Whether to issue a warning when the <code>NDData</code> unit attribute is changed from a non-None
<code>warn_unsupported_correlated</code>	Whether to issue a warning if <code>NDData</code> arithmetic is performed with uncertainties and the

Attributes Documentation

warn_setting_unit_directly
 Whether to issue a warning when the `NDData` unit attribute is changed from a non-None value to another value that data values/uncertainties are not scaled with the unit change.

warn_unsupported_correlated

Whether to issue a warning if `NDData` arithmetic is performed with uncertainties and the uncertainties do not support the propagation of correlated uncertainties.

FlagCollection

class `astropy.nddata.FlagCollection` (*args, **kwargs)

Bases: `collections.OrderedDict`

The purpose of this class is to provide a dictionary for containing arrays of flags for the `NDData` class. Flags should be stored in Numpy arrays that have the same dimensions as the parent data, so the `FlagCollection` class adds shape checking to an ordered dictionary class.

The `FlagCollection` should be initialized like an `OrderedDict`, but with the addition of a `shape=` keyword argument used to pass the `NDData` shape.

IncompatibleUncertaintiesException

exception `astropy.nddata.IncompatibleUncertaintiesException`

This exception should be used to indicate cases in which uncertainties with two different classes can not be propagated.

MissingDataAssociationException

exception `astropy.nddata.MissingDataAssociationException`

This exception should be used to indicate that an uncertainty instance has not been associated with a parent `NDData` object.

NDData

class `astropy.nddata.NDData` (data, uncertainty=None, mask=None, flags=None, wcs=None, meta=None, unit=None)

Bases: `object`

A Superclass for array-based data in Astropy.

The key distinction from raw numpy arrays is the presence of additional metadata such as uncertainties, a mask, units, flags, and/or a coordinate system.

Parameters

data : `ndarray` or `NDData`

The actual data contained in this `NDData` object. Not that this will always be copied by *reference*, so you should make copy the `data` before passing it in if that's the desired behavior.

uncertainty : `NDUncertainty`, optional

Uncertainties on the data.

mask : `ndarray`-like, optional

Mask for the data, given as a boolean Numpy array or any object that can be converted to a boolean Numpy array with a shape matching that of the data. The values must be `False` where the data is *valid* and `True` when it is not (like Numpy masked arrays).

If `data` is a numpy masked array, providing `mask` here will causes the mask from the masked array to be ignored.

flags : ndarray-like or `FlagCollection`, optional

Flags giving information about each pixel. These can be specified either as a Numpy array of any type (or an object which can be converted to a Numpy array) with a shape matching that of the data, or as a `FlagCollection` instance which has a shape matching that of the data.

wcs : undefined, optional

WCS-object containing the world coordinate system for the data.

Warning: This is not yet defined because the discussion of how best to represent this class's WCS system generically is still under consideration. For now just leave it as `None`

meta : dict-like object, optional

Metadata for this object. "Metadata" here means all information that is included with this object but not part of any other attribute of this particular object. e.g., creation date, unique identifier, simulation parameters, exposure time, telescope name, etc.

unit : `UnitBase` instance or str, optional

The units of the data.

Raises

ValueError :

If the `uncertainty` or `mask` inputs cannot be broadcast (e.g., match shape) onto data.

Notes

`NDData` objects can be easily converted to a regular Numpy array using `numpy.asarray`

For example:

```
>>> from astropy.nddata import NDData
>>> import numpy as np
>>> x = NDData([1,2,3])
>>> np.asarray(x)
array([1, 2, 3])
```

If the `NDData` object has a `mask`, `numpy.asarray` will return a Numpy masked array.

This is useful, for example, when plotting a 2D image using `matplotlib`:

```
>>> from astropy.nddata import NDData
>>> from matplotlib import pyplot as plt
>>> x = NDData([[1,2,3], [4,5,6]])
>>> plt.imshow(x)
```

Attributes Summary

<code>dtype</code>	<code>numpy.dtype</code> of this object's data.
--------------------	---

Continued on next page

Table 7.3 – continued from previous page

<code>flags</code>	
<code>mask</code>	
<code>ndim</code>	integer dimensions of this object's data
<code>shape</code>	shape tuple of this object's data.
<code>size</code>	integer size of this object's data.
<code>uncertainty</code>	
<code>unit</code>	

Methods Summary

<code>add(operand[, propagate_uncertainties])</code>	Add another dataset (<code>operand</code>) to this dataset.
<code>convert_unit_to(unit[, equivalencies])</code>	Returns a new <code>NDData</code> object whose values have been converted to a new unit.
<code>divide(operand[, propagate_uncertainties])</code>	Divide another dataset (<code>operand</code>) to this dataset.
<code>multiply(operand[, propagate_uncertainties])</code>	Multiply another dataset (<code>operand</code>) to this dataset.
<code>read(*args, **kwargs)</code>	Read in data
<code>subtract(operand[, propagate_uncertainties])</code>	Subtract another dataset (<code>operand</code>) to this dataset.
<code>write(data, *args, **kwargs)</code>	Write out data

Attributes Documentation

dtype

`numpy.dtype` of this object's data.

flags

mask

ndim

integer dimensions of this object's data

shape

shape tuple of this object's data.

size

integer size of this object's data.

uncertainty

unit

Methods Documentation

add (*operand*, *propagate_uncertainties=True*)

Add another dataset (*operand*) to this dataset.

Parameters

operand : `NDData`

The second operand in the operation $a + b$

propagate_uncertainties : bool

Whether to propagate uncertainties following the propagation rules defined by the class used for the `uncertainty` attribute.

Returns

result : `NDData`

The resulting dataset

Notes

This method requires the datasets to have identical WCS properties, equivalent units, and identical shapes. Flags and meta-data get set to None in the resulting dataset. The unit in the result is the same as the unit in `self`. Uncertainties are propagated, although correlated errors are not supported by any of the built-in uncertainty classes. If uncertainties are assumed to be correlated, a warning is issued by default (though this can be disabled via the `astropy.nddata.conf.warn_unsupported_correlated` configuration item). Values masked in either dataset before the operation are masked in the resulting dataset.

convert_unit_to (*unit*, *equivalencies*=[])

Returns a new `NDData` object whose values have been converted to a new unit.

Parameters

unit : `astropy.units.UnitBase` instance or str

The unit to convert to.

equivalencies : list of equivalence pairs, optional

A list of equivalence pairs to try if the units are not directly convertible. See *Equivalencies*.

Returns

result : `NDData`

The resulting dataset

Raises

UnitsError

If units are inconsistent.

Notes

Flags are set to None in the result.

divide (*operand*, *propagate_uncertainties*=True)

Divide another dataset (*operand*) to this dataset.

Parameters

operand : `NDData`

The second operand in the operation a / b

propagate_uncertainties : bool

Whether to propagate uncertainties following the propagation rules defined by the class used for the `uncertainty` attribute.

Returns

result : `NDData`

The resulting dataset

Notes

This method requires the datasets to have identical WCS properties, equivalent units, and identical shapes. Flags and meta-data get set to `None` in the resulting dataset. The unit in the result is the same as the unit in `self`. Uncertainties are propagated, although correlated errors are not supported by any of the built-in uncertainty classes. If uncertainties are assumed to be correlated, a warning is issued by default (though this can be disabled via the `astropy.nddata.conf.warn_unsupported_correlated` configuration item). Values masked in either dataset before the operation are masked in the resulting dataset.

multiply (*operand*, *propagate_uncertainties=True*)

Multiply another dataset (*operand*) to this dataset.

Parameters

operand : `NDData`

The second operand in the operation $a * b$

propagate_uncertainties : bool

Whether to propagate uncertainties following the propagation rules defined by the class used for the `uncertainty` attribute.

Returns

result : `NDData`

The resulting dataset

Notes

This method requires the datasets to have identical WCS properties, equivalent units, and identical shapes. Flags and meta-data get set to `None` in the resulting dataset. The unit in the result is the same as the unit in `self`. Uncertainties are propagated, although correlated errors are not supported by any of the built-in uncertainty classes. If uncertainties are assumed to be correlated, a warning is issued by default (though this can be disabled via the `astropy.nddata.conf.warn_unsupported_correlated` configuration item). Values masked in either dataset before the operation are masked in the resulting dataset.

classmethod read (**args*, ***kwargs*)

Read in data

The arguments passed to this method depend on the format

subtract (*operand*, *propagate_uncertainties=True*)

Subtract another dataset (*operand*) to this dataset.

Parameters

operand : `NDData`

The second operand in the operation $a - b$

propagate_uncertainties : bool

Whether to propagate uncertainties following the propagation rules defined by the class used for the `uncertainty` attribute.

Returns**result** : `NDData`

The resulting dataset

Notes

This method requires the datasets to have identical WCS properties, equivalent units, and identical shapes. Flags and meta-data get set to `None` in the resulting dataset. The unit in the result is the same as the unit in `self`. Uncertainties are propagated, although correlated errors are not supported by any of the built-in uncertainty classes. If uncertainties are assumed to be correlated, a warning is issued by default (though this can be disabled via the `astropy.nddata.conf.warn_unsupported_correlated` configuration item). Values masked in either dataset before the operation are masked in the resulting dataset.

write (*data*, **args*, ***kwargs*)

Write out data

The arguments passed to this method depend on the format

NDUncertainty**class** `astropy.nddata.NDUncertainty`Bases: `object`

This is the base class for uncertainty classes used with `NDData`. It is implemented as an abstract class and should never be directly instantiated.

Classes inheriting from `NDData` should overload the `propagate_*` methods, keeping the call signature the same. The propagate methods can assume that a `parent_nddata` attribute is present which links to the parent `nddata` dataset, and take an `NDData` instance as the positional argument, *not* an `NDUncertainty` instance, because the `NDData` instance can be used to access both the data and the uncertainties (some propagations require the data values).

Attributes Summary

<code>parent_nddata</code>	
<code>supports_correlated</code>	<code>bool(x) -> bool</code>

Methods Summary

<code>propagate_add(other_nddata, result_data)</code>	Propagate uncertainties for addition.
<code>propagate_divide(other_nddata, result_data)</code>	Propagate uncertainties for division.
<code>propagate_multiply(other_nddata, result_data)</code>	Propagate uncertainties for multiplication.
<code>propagate_subtract(other_nddata, result_data)</code>	Propagate uncertainties for subtraction.

Attributes Documentation**parent_nddata**

`supports_correlated = False`

Methods Documentation

`propagate_add` (*other_nddata*, *result_data*)

Propagate uncertainties for addition.

Parameters

other_nddata : NDData instance

The data for the second other_nddata in a + b

result_data : ndarray instance

The data array that is the result of the addition

Returns

result_uncertainty : NDUncertainty instance

The resulting uncertainty

Raises

IncompatibleUncertaintiesException

Raised if the method does not know how to add the uncertainties

`propagate_divide` (*other_nddata*, *result_data*)

Propagate uncertainties for division.

Parameters

other_nddata : NDData instance

The data for the second other_nddata in a + b

result_data : ndarray instance

The data array that is the result of the addition

Returns

result_uncertainty : NDUncertainty instance

The resulting uncertainty

`propagate_multiply` (*other_nddata*, *result_data*)

Propagate uncertainties for multiplication.

Parameters

other_nddata : NDData instance

The data for the second other_nddata in a + b

result_data : ndarray instance

The data array that is the result of the addition

Returns

result_uncertainty : NDUncertainty instance

The resulting uncertainty

`propagate_subtract` (*other_nddata*, *result_data*)

Propagate uncertainties for subtraction.

Parameters**other_nddata** : NDData instance

The data for the second other_nddata in a + b

result_data : ndarray instance

The data array that is the result of the addition

Returns**result_uncertainty** : NDUncertainty instance

The resulting uncertainty

Raises**IncompatibleUncertaintiesException**

Raised if the method does not know how to add the uncertainties

StdDevUncertainty**class** astropy.nddata.**StdDevUncertainty** (*array=None, copy=True*)

Bases: astropy.nddata.NDUncertainty

A class for standard deviation uncertainties

Attributes Summary

array	
parent_nddata	
support_correlated	bool(x) -> bool

Methods Summary

propagate_add(other_nddata, result_data)	Propagate uncertainties for addition.
propagate_divide(other_nddata, result_data)	Propagate uncertainties for division.
propagate_multiply(other_nddata, result_data)	Propagate uncertainties for multiplication.
propagate_subtract(other_nddata, result_data)	Propagate uncertainties for subtraction.

Attributes Documentation**array****parent_nddata****support_correlated = False**

Methods Documentation

propagate_add (*other_nddata*, *result_data*)

Propagate uncertainties for addition.

Parameters

other_nddata : NDData instance

The data for the second other_nddata in a + b

result_data : ndarray instance

The data array that is the result of the addition

Returns

result_uncertainty : NDUncertainty instance

The resulting uncertainty

Raises

IncompatibleUncertaintiesException

Raised if the method does not know how to propagate the uncertainties

propagate_divide (*other_nddata*, *result_data*)

Propagate uncertainties for division.

Parameters

other_nddata : NDData instance

The data for the second other_nddata in a + b

result_data : ndarray instance

The data array that is the result of the addition

Returns

result_uncertainty : NDUncertainty instance

The resulting uncertainty

Raises

IncompatibleUncertaintiesException

Raised if the method does not know how to propagate the uncertainties

propagate_multiply (*other_nddata*, *result_data*)

Propagate uncertainties for multiplication.

Parameters

other_nddata : NDData instance

The data for the second other_nddata in a + b

result_data : ndarray instance

The data array that is the result of the addition

Returns

result_uncertainty : NDUncertainty instance

The resulting uncertainty

Raises

IncompatibleUncertaintiesException

Raised if the method does not know how to propagate the uncertainties

propagate_subtract (*other_nddata*, *result_data*)

Propagate uncertainties for subtraction.

Parameters

other_nddata : NDData instance

The data for the second other_nddata in a + b

result_data : ndarray instance

The data array that is the result of the addition

Returns

result_uncertainty : NDUncertainty instance

The resulting uncertainty

Raises

IncompatibleUncertaintiesException

Raised if the method does not know how to propagate the uncertainties

DATA TABLES (ASTROPY.TABLE)

8.1 Introduction

`astropy.table` provides functionality for storing and manipulating heterogeneous tables of data in a way that is familiar to `numpy` users. A few notable features of this package are:

- Initialize a table from a wide variety of input data structures and types.
- Modify a table by adding or removing columns, changing column names, or adding new rows of data.
- Handle tables containing missing values.
- Include table and column metadata as flexible data structures.
- Specify a description, units and output formatting for columns.
- Interactively scroll through long tables similar to using `more`.
- Create a new table by selecting rows or columns from a table.
- Perform *Table operations* like database joins and concatenation.
- Manipulate multidimensional columns.
- Methods for *Reading and writing Table objects* to files
- Hooks for *Subclassing Table* and its component classes

Currently `astropy.table` is used when reading an ASCII table using `astropy.io.ascii`. Future releases of AstroPy are expected to use the `Table` class for other subpackages such as `astropy.io.votable` and `astropy.io.fits`.

8.2 Getting Started

The basic workflow for creating a table, accessing table elements, and modifying the table is shown below. These examples show a very simple case, while the full `astropy.table` documentation is available from the *Using table* section.

First create a simple table with three columns of data named `a`, `b`, and `c`. These columns have integer, float, and string values respectively:

```
>>> from astropy.table import Table
>>> a = [1, 4, 5]
>>> b = [2.0, 5.0, 8.2]
>>> c = ['x', 'y', 'z']
>>> t = Table([a, b, c], names=('a', 'b', 'c'), meta={'name': 'first table'})
```

If you have row-oriented input data such as a list of records, use the `rows` keyword:

```
>>> data_rows = [(1, 2.0, 'x'),
...              (4, 5.0, 'y'),
...              (5, 8.2, 'z')]
>>> t = Table(rows=data_rows, names=('a', 'b', 'c'), meta={'name': 'first table'})
```

There are a few ways to examine the table. You can get detailed information about the table values and column definitions as follows:

```
>>> t
<Table rows=3 names=('a', 'b', 'c')>
array([(1, 2.0, 'x'), (4, 5.0, 'y'), (5, 8.2, 'z')],
      dtype=[('a', '<i8'), ('b', '<f8'), ('c', 'S1')])
```

One can also assign an unit to the columns. If any column has an unit assigned, all units would be shown as follows:

```
>>> t['b'].unit = 's'
>>> t
<Table rows=3 names=('a', 'b', 'c') units=(None, 's', None)>
array([(1, 2.0, 'x'), (4, 5.0, 'y'), (5, 8.2, 'z')],
      dtype=[('a', '<i8'), ('b', '<f8'), ('c', 'S1')])
```

From within the IPython notebook, the table is displayed as a formatted HTML table:

In [19]: `t`

Out[19]:

a	b	c
1	2.0	x
4	5.0	y
5	8.2	z

If you print the table (either from the notebook or in a text console session) then a formatted version appears:

```
>>> print(t)
  a   b   c
  ---
  1 2.0  x
  4 5.0  y
  5 8.2  z
```

If you do not like the format of a particular column, you can change it:

```
>>> t['b'].format = '7.3f'
>>> print(t)
  a     b     c
  ---
  1  2.000  x
  4  5.000  y
  5  8.200  z
```

For a long table you can scroll up and down through the table one page at time:

```
>>> t.more()
```

You can also display it as an HTML-formatted table in the browser:

```
>>> t.show_in_browser()
```

or as an interactive (searchable & sortable) javascript table:

```
>>> t.show_in_browser(jsviewer=True)
```

Now examine some high-level information about the table:

```
>>> t.colnames
['a', 'b', 'c']
>>> len(t)
3
>>> t.meta
{'name': 'first table'}
```

Access the data by column or row using familiar `numpy` structured array syntax:

```
>>> t['a']          # Column 'a'
<Column name='a' unit=None format=None description=None>
array([1, 4, 5])

>>> t['a'][1]      # Row 1 of column 'a'
4

>>> t[1]          # Row obj for with row 1 values
<Row 1 of table
  values=(4, 5.0, 'y')
  dtype=[('a', '<i8'), ('b', '<f8'), ('c', 'S1')]>

>>> t[1]['a']     # Column 'a' of row 1
4
```

One can retrieve a subset of a table by rows (using a slice) or columns (using column names), where the subset is returned as a new table:

```
>>> print(t[0:2])      # Table object with rows 0 and 1
  a      b      c
  s
---
  1  2.000  x
  4  5.000  y

>>> print(t['a', 'c']) # Table with cols 'a', 'c'
  a      c
  ---
  1      x
  4      y
  5      z
```

Modifying table values in place is flexible and works as one would expect:

```
>>> t['a'] = [-1, -2, -3]      # Set all column values
>>> t['a'][2] = 30             # Set row 2 of column 'a'
>>> t[1] = (8, 9.0, "W")      # Set all row values
>>> t[1]['b'] = -9            # Set column 'b' of row 1
>>> t[0:2]['b'] = 100.0       # Set column 'b' of rows 0 and 1
>>> print(t)
  a      b      c
  s
---
  1      x
  4      y
  5      z
```

```
-1 100.000  x
 8 100.000  W
30  8.200   z
```

Add, remove, and rename columns with the following:

```
>>> t['d'] = [1, 2, 3]
>>> del t['c']
>>> t.rename_column('a', 'A')
>>> t.colnames
['A', 'b', 'd']
```

Adding a new row of data to the table is as follows:

```
>>> t.add_row([-8, -9, 10])
>>> len(t)
4
```

Lastly, one can create a table with support for missing values, for example by setting `masked=True`:

```
>>> t = Table([a, b, c], names=('a', 'b', 'c'), masked=True)
>>> t['a'].mask = [True, True, False]
>>> t
<Table rows=3 names=('a','b','c')>
masked_array(data = [(--, 2.0, 'x') (--, 5.0, 'y') (5, 8..., 'z')],
             mask = [(True, False, False) (True, False, False) (False, False, False)],
             fill_value = (999999, 1e+20, 'N'),
             dtype = [('a', '<i8'), ('b', '<f8'), ('c', 'S1')])

>>> print(t)
  a    b    c
--- --- ---
-- 2.0  x
-- 5.0  y
 5 8.2  z
```

8.3 Using `table`

The details of using `astropy.table` are provided in the following sections:

8.3.1 Construct table

Constructing a table

There is great deal of flexibility in the way that a table can be initially constructed. Details on the inputs to the `Table` constructor are in the [Initialization Details](#) section. However, the easiest way to understand how to make a table is by example.

Examples

Much of the flexibility lies in the types of data structures which can be used to initialize the table data. The examples below show how to create a table from scratch with no initial data, create a table with a list of columns, a dictionary of columns, or from `numpy` arrays (either structured or homogeneous).

Setup For the following examples you need to import the `Table` and `Column` classes along with the `numpy` package:

```
>>> from astropy.table import Table, Column
>>> import numpy as np
```

Creating from scratch A `Table` can be created without any initial input data or even without any initial columns. This is useful for building tables dynamically if the initial size, columns, or data are not known.

Note: Adding columns or rows requires making a new copy of the entire table each time, so in the case of large tables this may be slow.

```
>>> t = Table()
>>> t['a'] = [1, 4]
>>> t['b'] = Column([2.0, 5.0], unit='cm', description='Velocity')
>>> t['c'] = ['x', 'y']

>>> t = Table(names=('a', 'b', 'c'), dtype=('f4', 'i4', 'S2'))
>>> t.add_row((1, 2.0, 'x'))
>>> t.add_row((4, 5.0, 'y'))
```

List of columns A typical case is where you have a number of data columns with the same length defined in different variables. These might be Python lists or `numpy` arrays or a mix of the two. These can be used to create a `Table` by putting the column data variables into a Python list. In this case the column names are not defined by the input data, so they must either be set using the `names` keyword or they will be auto-generated as `col<N>`.

```
>>> a = [1, 4]
>>> b = [2.0, 5.0]
>>> c = ['x', 'y']
>>> t = Table([a, b, c], names=('a', 'b', 'c'))
>>> t
<Table rows=2 names=('a', 'b', 'c')>
array([(1, 2.0, 'x'), (4, 5.0, 'y')],
      dtype=[('a', '<i8'), ('b', '<f8'), ('c', 'S1')])
```

Make a new table using columns from the first table

Once you have a `Table` then you can make new table by selecting columns and putting this into a Python list, e.g. [`t['c']`, `t['a']`]:

```
>>> Table([t['c'], t['a']])
<Table rows=2 names=('c', 'a')>
array([('x', 1), ('y', 4)],
      dtype=[('c', 'S1'), ('a', '<i8')])
```

Make a new table using expressions involving columns

The `Column` object is derived from the standard `numpy` array and can be used directly in arithmetic expressions. This allows for a compact way of making a new table with modified column values:

```
>>> Table([t['a']**2, t['b'] + 10])
<Table rows=2 names=('a', 'b')>
array([(1, 12.0), (16, 15.0)],
      dtype=[('a', '<i8'), ('b', '<f8')])
```

Different types of column data

The list input method for `Table` is very flexible since you can use a mix of different data types to initialize a table:

```
>>> a = (1, 4)
>>> b = np.array([[2, 3], [5, 6]]) # vector column
>>> c = Column(['x', 'y'], name='axis')
>>> arr = (a, b, c)
>>> Table(arr) # Data column named "c" has a name "axis" that table
<Table rows=2 names=('col0', 'col1', 'axis')>
array([(1, [2, 3], 'x'), (4, [5, 6], 'y')],
      dtype=[('col0', '<i8'), ('col1', '<i8', (2,)), ('axis', 'S1')])
```

Notice that in the third column the existing column name `'axis'` is used.

Dict of columns A dictionary of column data can be used to initialize a `Table`.

```
>>> arr = {'a': [1, 4],
...       'b': [2.0, 5.0],
...       'c': ['x', 'y']}
>>>
>>> Table(arr)
<Table rows=2 names=('a', 'c', 'b')>
array([(1, 'x', 2.0), (4, 'y', 5.0)],
      dtype=[('a', '<i8'), ('c', 'S1'), ('b', '<f8')])
```

Specify the column order and optionally the data types

```
>>> Table(arr, names=('a', 'b', 'c'), dtype=('f4', 'i4', 'S2'))
<Table rows=2 names=('a', 'b', 'c')>
array([(1.0, 2, 'x'), (4.0, 5, 'y')],
      dtype=[('a', '<f4'), ('b', '<i4'), ('c', 'S2')])
```

Different types of column data

The input column data can be any data type that can initialize a `Column` object:

```
>>> arr = {'a': (1, 4),
...       'b': np.array([[2, 3], [5, 6]]),
...       'c': Column(['x', 'y'], name='axis')}
>>> Table(arr, names=('a', 'b', 'c'))
<Table rows=2 names=('a', 'b', 'c')>
array([(1, [2, 3], 'x'), (4, [5, 6], 'y')],
      dtype=[('a', '<i8'), ('b', '<i8', (2,)), ('c', 'S1')])
```

Notice that the key `'c'` takes precedence over the existing column name `'axis'` in the third column. Also see that the `'b'` column is a vector column where each row element is itself a 2-element array.

Renaming columns is not possible

```
>>> Table(arr, names=('a_new', 'b_new', 'c_new'))
Traceback (most recent call last):
...
KeyError: 'a_new'
Traceback (most recent call last):
...
KeyError: 'a_new'
```

Row data Row-oriented data can be used to create a table using the `rows` keyword argument.

List of data records as list or tuple

If you have row-oriented input data such as a list of records, you need to use the `rows` keyword to create a table:

```
>>> data_rows = [(1, 2.0, 'x'),
...              (4, 5.0, 'y'),
...              (5, 8.2, 'z')]
>>> t = Table(rows=data_rows, names=('a', 'b', 'c'))
>>> print(t)
  a   b   c
--- --- ---
  1  2.0  x
  4  5.0  y
  5  8.2  z
```

The data object passed as the `rows` argument can be any form which is parsable by the `np.rec.fromrecords()` function.

List of dict objects

You can also initialize a table with row values. This is constructed as a list of dict objects. The keys determine the column names:

```
>>> data = [{'a': 5, 'b': 10},
...         {'a': 15, 'b': 20}]
>>> Table(rows=data)
<Table rows=2 names=('a', 'b')>
array([(5, 10), (15, 20)],
      dtype=[('a', '<i8'), ('b', '<i8')])
```

Every row must have the same set of keys or a `ValueError` will be thrown:

```
>>> t = Table(rows=[{'a': 5, 'b': 10}, {'a': 15, 'b': 30, 'c': 50}])
Traceback (most recent call last):
...
ValueError: Row 0 has no value for column c
Traceback (most recent call last):
...
ValueError: Row 0 has no value for column c
```

Single row

You can also make a new table from a single row of an existing table:

```
>>> a = [1, 4]
>>> b = [2.0, 5.0]
>>> t = Table([a, b], names=('a', 'b'))
>>> t2 = Table(rows=t[1])
```

Remember that a `Row` has effectively a zero length compared to the newly created `Table` which has a length of one. This is similar to the difference between a scalar `1` (length 0) and an array like `np.array([1])` with length 1.

Note: In the case of input data as a list of dicts or a single `Table` row, it is allowed to supply the data as the `data` argument since these forms are always unambiguous. For example `Table({'a': 1}, {'a': 2})` is accepted. However, a list of records must always be provided using the `rows` keyword, otherwise it will be interpreted as a list of columns.

NumPy structured array The structured array is the standard mechanism in `numpy` for storing heterogenous table data. Most scientific I/O packages that read table files (e.g. `PyFITS`, `vo.table`, `asciitable`) will return the table in an object that is based on the structured array. A structured array can be created using:

```
>>> arr = np.array([(1, 2.0, 'x'),
...                 (4, 5.0, 'y')],
...                 dtype=[('a', 'i8'), ('b', 'f8'), ('c', 'S2')])
```

From `arr` it is simple to create the corresponding `Table` object:

```
>>> Table(arr)
<Table rows=2 names=('a','b','c')>
array([(1, 2.0, 'x'), (4, 5.0, 'y')],
      dtype=[('a', '<i8'), ('b', '<f8'), ('c', 'S2')])
```

Note that in the above example and most the following ones we are creating a table and immediately asking the interactive Python interpreter to print the table to see what we made. In real code you might do something like:

```
>>> table = Table(arr)
>>> print table
 a  b  c
---  ---  ---
  1 2.0  x
  4 5.0  y
```

New column names

The column names can be changed from the original values by providing the `names` argument:

```
>>> Table(arr, names=('a_new', 'b_new', 'c_new'))
<Table rows=2 names=('a_new','b_new','c_new')>
array([(1, 2.0, 'x'), (4, 5.0, 'y')],
      dtype=[('a_new', '<i8'), ('b_new', '<f8'), ('c_new', 'S2')])
```

New data types

Likewise the data type for each column can be changed with `dtype`:

```
>>> Table(arr, dtype=('f4', 'i4', 'S4'))
<Table rows=2 names=('a','b','c')>
array([(1.0, 2, 'x'), (4.0, 5, 'y')],
      dtype=[('a', '<f4'), ('b', '<i4'), ('c', 'S4')])

>>> Table(arr, names=('a_new', 'b_new', 'c_new'), dtype=('f4', 'i4', 'S4'))
<Table rows=2 names=('a_new','b_new','c_new')>
array([(1.0, 2, 'x'), (4.0, 5, 'y')],
      dtype=[('a_new', '<f4'), ('b_new', '<i4'), ('c_new', 'S4')])
```

NumPy homogeneous array A normal `numpy` 2-d array (where all elements have the same type) can be converted into a `Table`. In this case the column names are not specified by the data and must either be provided by the user or will be automatically generated as `col<N>` where `<N>` is the column number.

Basic example with automatic column names

```
>>> arr = np.array([[1, 2, 3],
...                 [4, 5, 6]])
>>> Table(arr)
<Table rows=2 names=('col0','col1','col2')>
array([(1, 2, 3), (4, 5, 6)],
      dtype=[('col0', '<i8'), ('col1', '<i8'), ('col2', '<i8')])
```

Column names and types specified

```
>>> Table(arr, names=('a_new', 'b_new', 'c_new'), dtype=('f4', 'i4', 'S4'))
<Table rows=2 names=('a_new', 'b_new', 'c_new')>
array([(1.0, 2, '3'), (4.0, 5, '6')],
      dtype=[('a_new', '<f4'), ('b_new', '<i4'), ('c_new', 'S4')])
```

Referencing the original data

It is possible to reference the original data for an homogeneous array as long as the data types are not changed:

```
>>> t = Table(arr, copy=False)
```

Python arrays versus ‘numpy’ arrays as input

There is a slightly subtle issue that is important to understand in the way that `Table` objects are created. Any data input that looks like a Python list (including a tuple) is considered to be a list of columns. In contrast an homogeneous `numpy` array input is interpreted as a list of rows:

```
>>> arr = [[1, 2, 3],
...        [4, 5, 6]]
>>> np_arr = np.array(arr)

>>> Table(arr)      # Two columns, three rows
<Table rows=3 names=('col0', 'col1')>
array([(1, 4), (2, 5), (3, 6)],
      dtype=[('col0', '<i8'), ('col1', '<i8')])

>>> Table(np_arr)  # Three columns, two rows
<Table rows=2 names=('col0', 'col1', 'col2')>
array([(1, 2, 3), (4, 5, 6)],
      dtype=[('col0', '<i8'), ('col1', '<i8'), ('col2', '<i8')])
```

This dichotomy is needed to support flexible list input while retaining the natural interpretation of 2-d `numpy` arrays where the first index corresponds to data “rows” and the second index corresponds to data “columns”.

Table columns A new table can be created by selecting a subset of columns in an existing table:

```
>>> t = Table(names=('a', 'b', 'c'))
>>> t2 = t['c', 'b', 'a'] # Makes a copy of the data
>>> print t2
  c  b  a
---  ---  ---
```

An alternate way to use the `columns` attribute (explained in the `TableColumns` section) to initialize a new table. This let’s you choose columns by their numerical index or name and supports slicing syntax:

```
>>> Table(t.columns[0:2])
<Table rows=0 names=('a', 'b')>
array([],
      dtype=[('a', '<f8'), ('b', '<f8')])

>>> Table([t.columns[0], t.columns['c']])
<Table rows=0 names=('a', 'c')>
array([],
      dtype=[('a', '<f8'), ('c', '<f8')])
```

Initialization Details

A table object is created by initializing a `Table` class object with the following arguments, all of which are optional:

data

[numpy ndarray, dict, list, or Table] Data to initialize table.

names

[list] Specify column names

dtype

[list] Specify column data types

meta

[dict-like] Meta-Data associated with the table

copy

[boolean] Copy the input data (default=True).

The following subsections provide further detail on the values and options for each of the keyword arguments that can be used to create a new `Table` object.

data The `Table` object can be initialized with several different forms for the `data` argument.

numpy ndarray (structured array)

The base column names are the field names of the `data` structured array. The `names` list (optional) can be used to select particular fields and/or reorder the base names. The `dtype` list (optional) must match the length of `names` and is used to override the existing `data` types.

numpy ndarray (homogeneous)

The `data` ndarray must be at least 2-dimensional, with the first (left-most) index corresponding to row number (table length) and the second index corresponding to column number (table width). Higher dimensions get absorbed in the shape of each table cell.

If provided the `names` list must match the “width” of the `data` argument. The default for `names` is to auto-generate column names in the form “col<N>”. If provided the `dtype` list overrides the base column types and must match the length of `names`.

dict-like

The keys of the `data` object define the base column names. The corresponding values can be `Column` objects, numpy arrays, or list-like objects. The `names` list (optional) can be used to select particular fields and/or reorder the base names. The `dtype` list (optional) must match the length of `names` and is used to override the existing or default data types.

list-like

Each item in the `data` list provides a column of data values and can be a `Column` object, numpy array, or list-like object. The `names` list defines the name of each column. The names will be auto-generated if not provided (either from the `names` argument or by `Column` objects). If provided the `names` argument must match the number of items in the `data` list. The optional `dtype` list will override the existing or default data types and must match `names` in length.

list-of-dicts

Similar to Python’s builtin `csv.DictReader`, each item in the `data` list provides a row of data values and must be a dict. The key values in each dict define the column names and each row must have identical column names. The `names` argument may be supplied to specify column ordering. If it is not provided, the column order will default to alphabetical. The `dtype` list may be specified, and must correspond to the order of output columns. If any row’s keys do not match the rest of the rows, a `ValueError` will be thrown.

None

Initialize a zero-length table. If `names` and optionally `dtype` are provided then the corresponding columns are created.

names The `names` argument provides a way to specify the table column names or override the existing ones. By default the column names are either taken from existing names (for `ndarray` or `Table` input) or auto-generated as `col<N>`. If `names` is provided then it must be a list with the same length as the number of columns. Any list elements with value `None` fall back to the default name.

In the case where `data` is provided as dict of columns, the `names` argument can be supplied to specify the order of columns. The `names` list must then contain each of the keys in the `data` dict. If `names` is not supplied then the order of columns in the output table is not determinate.

dtype The `dtype` argument provides a way to specify the table column data types or override the existing types. By default the types are either taken from existing types (for `ndarray` or `Table` input) or auto-generated by the `numpy.array()` routine. If `dtype` is provided then it must be a list with the same length as the number of columns. The values must be valid `numpy.dtype` initializers or `None`. Any list elements with value `None` fall back to the default type.

In the case where `data` is provided as dict of columns, the `dtype` argument must be accompanied by a corresponding `names` argument in order to uniquely specify the column ordering.

meta The `meta` argument is simply an object that contains meta-data associated with the table. It is recommended that this object be a dict or `OrderedDict`, but the only firm requirement is that it can be copied with the standard library `copy.deepcopy()` routine. By default `meta` is an empty `OrderedDict`.

copy By default the input `data` are copied into a new internal `np.ndarray` object in the `Table` object. In the case where `data` is either an `np.ndarray` object or an existing `Table`, it is possible to use a reference to the existing data by setting `copy=False`. This has the advantage of reducing memory use and being faster. However one should take care because any modifications to the new `Table` data will also be seen in the original input data. See the [Copy versus Reference](#) section for more information.

Copy versus Reference

Normally when a new `Table` object is created, the input data are *copied* into a new internal array object. This ensures that if the new table elements are modified then the original data will not be affected. However, when creating a table from a `numpy.ndarray` object (structured or homogeneous), it is possible to disable copying so that instead a memory reference to the original data is used. This has the advantage of being faster and using less memory. However, caution must be exercised because the new table data and original data will be linked, as shown below:

```
>>> arr = np.array([(1, 2.0, 'x'),
...                (4, 5.0, 'y')],
...                dtype=[('a', 'i8'), ('b', 'f8'), ('c', 'S2')])
>>> print arr['a'] # column "a" of the input array
[1 4]
>>> t = Table(arr, copy=False)
>>> t['a'][1] = 99
>>> print arr['a'] # arr['a'] got changed when we modified t['a']
[ 1 99]
```

Note that when referencing the data it is not possible to change the data types since that operation requires making a copy of the data. In this case an error occurs:

```
>>> t = Table(arr, copy=False, dtype=('f4', 'i4', 'S4'))
Traceback (most recent call last):
...
ValueError: Cannot specify dtype when copy=False
Traceback (most recent call last):
```

```
...
ValueError: Cannot specify dtype when copy=False
```

Another caveat in using referenced data is that you cannot add new row to the table. This generates an error because of conflict between the two references to the same underlying memory. Internally, adding a row may involve moving the data to a new memory location which would corrupt the input data object. `numpy` does not allow this:

```
>>> t.add_row([1, 2, 3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "astropy/table/table.py", line 760, in add_row
    self._data.resize((newlen,), refcheck=False)
ValueError: cannot resize this array: it does not own its data
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "astropy/table/table.py", line 760, in add_row
    self._data.resize((newlen,), refcheck=False)
ValueError: cannot resize this array: it does not own its data
```

Column and TableColumns classes

There are two classes, `Column` and `TableColumns`, that are useful when constructing new tables.

Column A `Column` object can be created as follows, where in all cases the column name should be provided as a keyword argument and one can optionally provide these values:

data

[list, ndarray or None] Column data values

dtype

[numpy.dtype compatible value] Data type for column

description

[str] Full description of column

unit

[str] Physical unit

format

[str or function] [Format specifier](#) for outputting column values

meta

[dict] Meta-data associated with the column

Initialization options The column data values, shape, and data type are specified in one of two ways:

Provide a “data“ value but not a “length“ or “shape“

Examples:

```
col = Column([1, 2], name='a') # shape=(2,)
col = Column([[1, 2], [3, 4]], name='a') # shape=(2, 2)
col = Column([1, 2], name='a', dtype=float)
col = Column(np.array([1, 2]), name='a')
col = Column(['hello', 'world'], name='a')
```

The `dtype` argument can be any value which is an acceptable fixed-size data-type initializer for the `numpy.dtype()` method. See <http://docs.scipy.org/doc/numpy/reference/arrays.dtypes.html>. Examples include:

- Python non-string type (float, int, bool)
- Numpy non-string type (e.g. `np.float32`, `np.int64`, `np.bool`)
- Numpy.dtype array-protocol type strings (e.g. `'i4'`, `'f8'`, `'S15'`)

If no `dtype` value is provided then the type is inferred using `np.array(data)`. When `data` is provided then the `shape` and `length` arguments are ignored.

Provide “length“ and optionally “shape“, but not “data“

Examples:

```
col = Column(name='a', length=5)
col = Column(name='a', dtype=int, length=10, shape=(3,4))
```

The default `dtype` is `np.float64`. The `shape` argument is the array shape of a single cell in the column. The default `shape` is `()` which means a single value in each element.

Note: After setting the type for a column, that type cannot be changed. If data values of a different type are assigned to the column then they will be cast to the existing column type.

Format specifier The format specifier controls the output of column values when a table or column is printed or written to an ASCII table. In the simplest case, it is a string that can be passed to python’s built-in `format` function. For more complicated formatting, one can also give “old-style” or “new-style” format strings, or even a function:

Plain format specification

This type of string specifies directly how the value should be formatted, using a [format specification mini-language](#) that is quite similar to C.

`" .4f"` will give four digits after the decimal in float format, or

`"6d"` will give integers in 6-character fields.

Old-style format string

This corresponds to syntax like `"%.4f" % value` as documented in [String formatting operations](#).

`"%.4f"` to print four digits after the decimal in float format, or

`"%6d"` to print an integer in a 6-character wide field.

New-style format string

This corresponds to syntax like `"{: .4f}".format(value)` as documented in [format string syntax](#).

`"{: .4f}"` to print four digits after the decimal in float format, or

`"{: 6d}"` to print an integer in a 6-character wide field.

Note that in either format string case any Python string that formats exactly one value is valid, so `{: .4f} angstroms` or `Value: %12.2f` would both work.

Function

The greatest flexibility can be achieved by setting a formatting function. This function must accept a single argument (the value) and return a string. In the following example this is used to make a LaTeX ready output:

```
>>> t = Table([[1,2],[1.234e9,2.34e-12]], names = ('a','b'))
>>> def latex_exp(value):
...     val = '{0:8.2}'.format(value)
...     mant, exp = val.split('e')
...     # remove leading zeros
...     exp = exp[0] + exp[1:].lstrip('0')
...     return '$ {0} \\times 10^{{{1}}}$'.format(mant, exp)
>>> t['b'].format = latex_exp
>>> t['a'].format = '.4f'
>>> import sys
>>> t.write(sys.stdout, format='latex')
\begin{table}
\begin{tabular}{cc}
a & b \\
1.0000 & $ 1.2 \times 10^{+9} $ \\
2.0000 & $ 2.3 \times 10^{-12} $ \\
\end{tabular}
\end{table}
```

TableColumns Each `Table` object has an attribute `columns` which is an ordered dictionary that stores all of the `Column` objects in the table (see also the `Column` section). Technically the `columns` attribute is a `TableColumns` object, which is an enhanced ordered dictionary that provides easier ways to select multiple columns. There are a few key points to remember:

- A `Table` can be initialized from a `TableColumns` object (copy is always `True`).
- Selecting multiple columns from a `TableColumns` object returns another `TableColumns` object.
- Select one column from a `TableColumns` object returns a `Column`.

So now look at the ways to select columns from a `TableColumns` object:

Select columns by name

```
>>> t = Table(names=('a', 'b', 'c', 'd'))

>>> t.columns['d', 'c', 'b']
<TableColumns names=('d', 'c', 'b')>
```

Select columns by index slicing

```
>>> t.columns[0:2] # Select first two columns
<TableColumns names=('a', 'b')>

>>> t.columns[::-1] # Reverse column order
<TableColumns names=('d', 'c', 'b', 'a')>
```

Select column by index or name

```
>>> t.columns[1] # Choose columns by index
<Column name='b' unit=None format=None description=None>
array([], dtype=float64)

>>> t.columns['b'] # Choose column by name
<Column name='b' unit=None format=None description=None>
array([], dtype=float64)
```

Subclassing Table

For some applications it can be useful to subclass the `Table` class in order to introduce specialized behavior. In addition to subclassing `Table` it is frequently desirable to change the behavior of the internal class objects which are contained or created by a `Table`. This includes rows, columns, formatting, and the columns container. In order to do this the subclass needs to declare what class to use (if it is different from the built-in version). This is done by specifying one or more of the class attributes `Row`, `Column`, `MaskedColumn`, `TableColumns`, or `TableFormatter`.

The following trivial example overrides all of these with do-nothing subclasses, but in practice you would override only the necessary subcomponents:

```
>>> from astropy.table import Table, Row, Column, MaskedColumn, TableColumns, TableFormatter

>>> class MyRow(Row): pass
>>> class MyColumn(Column): pass
>>> class MyMaskedColumn(MaskedColumn): pass
>>> class MyTableColumns(TableColumns): pass
>>> class MyTableFormatter(TableFormatter): pass

>>> class MyTable(Table):
...     """
...     Custom subclass of astropy.table.Table
...     """
...     Row = MyRow # Use MyRow to create a row object
...     Column = MyColumn # Column
...     MaskedColumn = MyMaskedColumn # Masked Column
...     TableColumns = MyTableColumns # Ordered dict holding Column objects
...     TableFormatter = MyTableFormatter # Controls table output
```

Example As a more practical example, suppose you have a table of data with a certain set of fixed columns, but you also want to carry an arbitrary dictionary of keyword=value parameters for each row and then access those values using the same item access syntax as if they were columns. It is assumed here that the extra parameters are contained in a numpy object-dtype column named `params`:

```
>>> from astropy.table import Table, Row
>>> class ParamsRow(Row):
...     """
...     Row class that allows access to an arbitrary dict of parameters
...     stored as a dict object in the ``params`` column.
...     """
...     def __getitem__(self, item):
...         if item not in self.colnames:
...             return self.data['params'][item]
...         else:
...             return self.data[item]
...
...     def keys(self):
...         out = [name for name in self.colnames if name != 'params']
...         params = [key.lower() for key in sorted(self.data['params'])]
...         return out + params
...
...     def values(self):
...         return [self[key] for key in self.keys()]
```

Now we put this into action with a trivial `Table` subclass:

```
>>> class ParamsTable(Table):
...     Row = ParamsRow
```

First make a table and add a couple of rows:

```
>>> t = ParamsTable(names=['a', 'b', 'params'], dtype=['i', 'f', 'O'])
>>> t.add_row((1, 2.0, {'x': 1.5, 'y': 2.5}))
>>> t.add_row((2, 3.0, {'z': 'hello', 'id': 123123}))
>>> print(t)
 a   b          params
-----
 1 2.0      {'y': 2.5, 'x': 1.5}
 2 3.0 {'z': 'hello', 'id': 123123}
```

Now see what we have from our specialized ParamsRow object:

```
>>> t[0]['y']
2.5
>>> t[1]['id']
123123
>>> t[1].keys()
['a', 'b', 'id', 'z']
>>> t[1].values()
[2, 3.0, 123123, 'hello']
```

To make this example really useful you might want to override `Table.__getitem__` in order to allow table-level access to the parameter fields. This might look something like:

```
class ParamsTable(table.Table):
    Row = ParamsRow

    def __getitem__(self, item):
        if isinstance(item, six.string_types):
            if item in self.colnames:
                return self.columns[item]
            else:
                # If item is not a column name then create a new MaskedArray
                # corresponding to self['params'][item] for each row. This
                # might not exist in some rows so mark as masked (missing) in
                # those cases.
                mask = np.zeros(len(self), dtype=np.bool)
                item = item.upper()
                values = [params.get(item) for params in self['params']]
                for ii, value in enumerate(values):
                    if value is None:
                        mask[ii] = True
                        values[ii] = ''
                return self.MaskedColumn(name=item, data=values, mask=mask)

        # ... and then the rest of the original __getitem__ ...
```

8.3.2 Access table

Accessing a table

Accessing the table properties and data is straightforward and is generally consistent with the basic interface for numpy structured arrays.

Quick overview

For the impatient, the code below shows the basics of accessing table data. Where relevant there is a comment about what sort of object. Except where noted, the table access returns objects that can be modified in order to update table data or properties. In cases where is returned and how the data contained in that object relate to the original table data (i.e. whether it is a copy or reference, see *Copy versus Reference*).

Make table

```
from astropy.table import Table
import numpy as np

arr = np.arange(15).reshape(5, 3)
t = Table(arr, names=('a', 'b', 'c'), meta={'keywords': {'key1': 'vall'}})
```

Table properties

```
t.columns # Dict of table columns
t.colnames # List of column names
t.meta # Dict of meta-data
len(t) # Number of table rows
```

Access table data

```
t['a'] # Column 'a'
t['a'][1] # Row 1 of column 'a'
t[1] # Row obj for with row 1 values
t[1]['a'] # Column 'a' of row 1
t[2:5] # Table object with rows 2:5
t[[1, 3, 4]] # Table object with rows 1, 3, 4 (copy)
t[np.array([1, 3, 4])] # Table object with rows 1, 3, 4 (copy)
t['a', 'c'] # Table with cols 'a', 'c' (copy)
dat = np.array(t) # Copy table data to numpy structured array object
```

Print table or column

```
print t # Print formatted version of table to the screen
t.pprint() # Same as above
t.pprint(show_unit=True) # Show column unit
t.pprint(show_name=False) # Do not show column names
t.pprint(max_lines=-1, max_width=-1) # Print full table no matter how long / wide it is

t.more() # Interactively scroll through table like Unix "more"

print t['a'] # Formatted column values
t['a'].pprint() # Same as above, with same options as Table.pprint()
t['a'].more() # Interactively scroll through column

lines = t.pformat() # Formatted table as a list of lines (same options as pprint)
lines = t['a'].pformat() # Formatted column values as a list
```

Details

For all the following examples it is assumed that the table has been created as below:

```
>>> from astropy.table import Table, Column
>>> import numpy as np
```

```
>>> arr = np.arange(15).reshape(5, 3)
>>> t = Table(arr, names=('a', 'b', 'c'), meta={'keywords': {'key1': 'vall'}})
>>> t['a'].format = "%6.3f" # print as a float with 3 digits after decimal point
>>> t['a'].unit = 'm sec^-1'
>>> t['a'].description = 'unladen swallow velocity'
>>> print t
  a      b  c
m sec^-1
-----
 0.000   1  2
 3.000   4  5
 6.000   7  8
 9.000  10 11
12.000  13 14
```

Accessing properties The code below shows accessing the table columns as a `TableColumns` object, getting the column names, table meta-data, and number of table rows. The table meta-data is simply an ordered dictionary (`OrderedDict`) by default.

```
>>> t.columns
<TableColumns names=('a', 'b', 'c')>

>>> t.colnames
['a', 'b', 'c']

>>> t.meta # Dict of meta-data
{'keywords': {'key1': 'vall'}}

>>> len(t)
5
```

Accessing data As expected one can access a table column by name and get an element from that column with a numerical index:

```
>>> t['a'] # Column 'a'
<Column name='a' unit='m sec^-1' format='%6.3f' description='unladen swallow velocity'>
array([ 0,  3,  6,  9, 12])

>>> t['a'][1] # Row 1 of column 'a'
3
```

When a table column is printed, either with `print` or via the `str()` built-in function, it is formatted according to the `format` attribute (see *Format specifier*):

```
>>> print(t['a'])
  a
-----
 0.000
 3.000
 6.000
 9.000
12.000
```

Likewise a table row and a column from that row can be selected:

```
>>> t[1] # Row object corresponding to row 1
<Row 1 of table
```

```
values=(3, 4, 5)
dtype=[('a', '<i4'), ('b', '<i8'), ('c', '<i8')]
```

```
>>> t[1]['a'] # Column 'a' of row 1
3
```

A Row object has the same columns and meta-data as its parent table:

```
>>> t[1].columns
<TableColumns names=('a','b','c')>

>>> t[1].colnames
['a', 'b', 'c']
```

Slicing a table returns a new table object which references to the original data within the slice region (See *Copy versus Reference*). The table meta-data and column definitions are copied.

```
>>> t[2:5] # Table object with rows 2:5 (reference)
<Table rows=3 names=('a','b','c') units=('m sec^-1',None,None)>
array([(6, 7, 8), (9, 10, 11), (12, 13, 14)],
      dtype=[('a', '<i8'), ('b', '<i8'), ('c', '<i8')])
```

It is possible to select table rows with an array of indexes or by specifying multiple column names. This returns a copy of the original table for the selected rows or columns.

```
>>> print t[[1, 3, 4]] # Table object with rows 1, 3, 4 (copy)
  a      b      c
m sec^-1
-----
  3.000   4   5
  9.000  10  11
 12.000  13  14
```

```
>>> print t[np.array([1, 3, 4])] # Table object with rows 1, 3, 4 (copy)
  a      b      c
m sec^-1
-----
  3.000   4   5
  9.000  10  11
 12.000  13  14
```

```
>>> print t['a', 'c'] # or t[['a', 'c']] or t[('a', 'c')]
... # Table with cols 'a', 'c' (copy)
  a      c
m sec^-1
-----
  0.000   2
  3.000   5
  6.000   8
  9.000  11
 12.000  14
```

Finally, one can access the underlying table data as a native `numpy` structured array by creating a copy or reference with `np.array`:

```
>>> data = np.array(t) # copy of data in t as a structured array
>>> data = np.array(t, copy=False) # reference to data in t
```

Formatted printing The values in a table or column can be printed or retrieved as a formatted table using one of several methods:

- `print` statement (Python 2) or `print()` function (Python 3).
- Table `more()` or Column `more()` methods to interactively scroll through table values.
- Table `pprint()` or Column `pprint()` methods to print a formatted version of the table to the screen.
- Table `pformat()` or Column `pformat()` methods to return the formatted table or column as a list of fixed-width strings. This could be used as a quick way to save a table.

These methods use *Format specifier* if available and strive to make the output readable. By default, table and column printing will not print the table larger than the available interactive screen size. If the screen size cannot be determined (in a non-interactive environment or on Windows) then a default size of 25 rows by 80 columns is used. If a table is too large then rows and/or columns are cut from the middle so it fits. For example:

```
>>> arr = np.arange(3000).reshape(100, 30) # 100 rows x 30 columns array
>>> t = Table(arr)
>>> print t
col0 col1 col2 col3 col4 col5 col6 ... col24 col25 col26 col27 col28 col29
---- ---- ---- ---- ---- ---- ---- ... ---- ---- ---- ---- ---- ----
    0    1    2    3    4    5    6 ...    24    25    26    27    28    29
   30   31   32   33   34   35   36 ...    54    55    56    57    58    59
   60   61   62   63   64   65   66 ...    84    85    86    87    88    89
   90   91   92   93   94   95   96 ...   114   115   116   117   118   119
  120  121  122  123  124  125  126 ...   144   145   146   147   148   149
  150  151  152  153  154  155  156 ...   174   175   176   177   178   179
  180  181  182  183  184  185  186 ...   204   205   206   207   208   209
  210  211  212  213  214  215  216 ...   234   235   236   237   238   239
  240  241  242  243  244  245  246 ...   264   265   266   267   268   269
  ...  ...  ...  ...  ...  ...  ... ...  ...  ...  ...  ...  ...  ...
2760 2761 2762 2763 2764 2765 2766 ... 2784 2785 2786 2787 2788 2789
2790 2791 2792 2793 2794 2795 2796 ... 2814 2815 2816 2817 2818 2819
2820 2821 2822 2823 2824 2825 2826 ... 2844 2845 2846 2847 2848 2849
2850 2851 2852 2853 2854 2855 2856 ... 2874 2875 2876 2877 2878 2879
2880 2881 2882 2883 2884 2885 2886 ... 2904 2905 2906 2907 2908 2909
2910 2911 2912 2913 2914 2915 2916 ... 2934 2935 2936 2937 2938 2939
2940 2941 2942 2943 2944 2945 2946 ... 2964 2965 2966 2967 2968 2969
2970 2971 2972 2973 2974 2975 2976 ... 2994 2995 2996 2997 2998 2999
```

more() method In order to browse all rows of a table or column use the Table `more()` or Column `more()` methods. These let you interactively scroll through the rows much like the linux `more` command. Once part of the table or column is displayed the supported navigation keys are:

f, space : forward one page
b : back one page
r : refresh same page
n : next row
p : previous row
< : go to beginning
> : go to end
q : quit browsing
h : print this help

pprint() method In order to fully control the print output use the Table `pprint()` or Column `pprint()` methods. These have keyword arguments `max_lines`, `max_width`, `show_name`, `show_unit` with meaning as shown below:

```
>>> arr = np.arange(3000, dtype=float).reshape(100, 30)
>>> t = Table(arr)
>>> t['col0'].format = '%e'
>>> t['col1'].format = '%.6f'
>>> t['col10'].unit = 'km**2'
>>> t['col29'].unit = 'kg sec m**-2'
```

```
>>> t.pprint(max_lines=8, max_width=40)
  col0      ...      col29
  km2      ... kg sec m**-2
----- ... -----
0.000000e+00 ...          29.0
3.000000e+01 ...          59.0
... ..
2.940000e+03 ...        2969.0
2.970000e+03 ...        2999.0
```

```
>>> t.pprint(max_lines=8, max_width=40, show_unit=True)
  col0      ...      col29
  km2      ... kg sec m**-2
----- ... -----
0.000000e+00 ...          29.0
3.000000e+01 ...          59.0
... ..
2.940000e+03 ...        2969.0
2.970000e+03 ...        2999.0
```

```
>>> t.pprint(max_lines=8, max_width=40, show_name=False)
  km2      ... kg sec m**-2
----- ... -----
0.000000e+00 ...          29.0
3.000000e+01 ...          59.0
6.000000e+01 ...          89.0
... ..
2.940000e+03 ...        2969.0
2.970000e+03 ...        2999.0
```

In order to force printing all values regardless of the output length or width set `max_lines` or `max_width` to `-1`, respectively. For the wide table in this example one sees 6 lines of wrapped output like the following:

```
>>> t.pprint(max_lines=6, max_width=-1)
  col0      col1      col2      col3      col4      col5      col6      col7      col8      col
9  col10     col11     col12     col13     col14     col15     col16     col17     col18     col20
  col21     col22     col23     col24     col25     col26     col27     col28     col29
-----
-----
-----
0.000000e+00  1.000000  2.0    3.0    4.0    5.0    6.0    7.0    8.0    9
.0   10.0  11.0  12.0  13.0  14.0  15.0  16.0  17.0  18.0  19.0  20.
0   21.0  22.0  23.0  24.0  25.0  26.0  27.0  28.0  29.0
3.000000e+01  31.000000  32.0   33.0   34.0   35.0   36.0   37.0   38.0   39
.0   40.0  41.0  42.0  43.0  44.0  45.0  46.0  47.0  48.0  49.0  50.
0   51.0  52.0  53.0  54.0  55.0  56.0  57.0  58.0  59.0
...      ...      ...      ...      ...      ...      ...      ...      ...      .
```

```
..      ...      ...      ...      ...      ...      ...      ...      ...      ..
.      ...      ...      ...      ...      ...      ...      ...      ...
2.970000e+03 2971.000000 2972.0 2973.0 2974.0 2975.0 2976.0 2977.0 2978.0 2979
.0 2980.0 2981.0 2982.0 2983.0 2984.0 2985.0 2986.0 2987.0 2988.0 2989.0 2990.
0 2991.0 2992.0 2993.0 2994.0 2995.0 2996.0 2997.0 2998.0 2999.0
```

For columns the syntax and behavior of `pprint()` is the same except that there is no `max_width` keyword argument:

```
>>> t['col3'].pprint(max_lines=8)
col3
-----
   3.0
  33.0
 63.0
   ...
2943.0
2973.0
```

pformat() method In order to get the formatted output for manipulation or writing to a file use the `Table.pformat()` or `Column.pformat()` methods. These behave just as for `pprint()` but return a list corresponding to each formatted line in the `pprint()` output.

```
>>> lines = t['col3'].pformat(max_lines=8)
>>> lines
[' col3 ', '-----', '   3.0', '  33.0', ' 63.0', '   ...', '2943.0', '2973.0']
```

Multidimensional columns If a column has more than one dimension then each element of the column is itself an array. In the example below there are 3 rows, each of which is a 2×2 array. The formatted output for such a column shows only the first and last value of each row element and indicates the array dimensions in the column name header:

```
>>> from astropy.table import Table, Column
>>> import numpy as np
>>> t = Table()
>>> arr = [ np.array([[ 1,  2],
...                  [10, 20]]),
...        np.array([[ 3,  4],
...                  [30, 40]]),
...        np.array([[ 5,  6],
...                  [50, 60]]) ]
>>> t['a'] = arr
>>> t['a'].shape
(3, 2, 2)
>>> t.pprint()
a [2,2]
-----
1 .. 20
3 .. 40
5 .. 60
```

In order to see all the data values for a multidimensional column use the column representation. This uses the standard `numpy` mechanism for printing any array:

```
>>> t['a']
<Column name='a' unit=None format=None description=None>
array([[ [ 1,  2],
         [10, 20]],
```

```
[[ 3,  4],
 [30, 40]],
 [[ 5,  6],
 [50, 60]]])
```

8.3.3 Modify table

Modifying a table

The data values within a `Table` object can be modified in much the same manner as for `numpy` structured arrays by accessing columns or rows of data and assigning values appropriately. A key enhancement provided by the `Table` class is the ability to easily modify the structure of the table: one can add or remove columns, and add new rows of data.

Quick overview

The code below shows the basics of modifying a table and its data.

Make a table

```
>>> from astropy.table import Table
>>> import numpy as np
>>> arr = np.arange(15).reshape(5, 3)
>>> t = Table(arr, names=('a', 'b', 'c'), meta={'keywords': {'key1': 'vall'}})
```

Modify data values

```
>>> t['a'] = [1, -2, 3, -4, 5] # Set all column values
>>> t['a'][2] = 30 # Set row 2 of column 'a'
>>> t[1] = (8, 9, 10) # Set all row values
>>> t[1]['b'] = -9 # Set column 'b' of row 1
>>> t[0:3]['c'] = 100 # Set column 'c' of rows 0, 1, 2
```

Note that `table[row][column]` assignments will not work with `numpy` “fancy” row indexing (in that case `table[row]` would be a *copy* instead of a *view*). “Fancy” `numpy` indices include a list, `numpy.ndarray`, or tuple of `numpy.ndarray` (e.g. the return from `numpy.where`):

```
>>> t[[1, 2]]['a'] = [3., 5.] # doesn't change table t
>>> t[np.array([1, 2])]['a'] = [3., 5.] # doesn't change table t
>>> t[np.where(t['a'] > 3)]['a'] = 3. # doesn't change table t
```

Instead use `table[column][row]` order:

```
>>> t['a'][[1, 2]] = [3., 5.]
>>> t['a'][np.array([1, 2])] = [3., 5.]
>>> t['a'][np.where(t['a'] > 3)] = 3.
```

Add a column or columns

A single column can be added to a table using syntax like adding a dict value. The value on the right hand side can be a list or array of the correct size, or a scalar value that will be broadcast:

```
>>> t['d1'] = np.arange(5)
>>> t['d2'] = [1, 2, 3, 4, 5]
>>> t['d3'] = 6 # all 5 rows set to 6
```

For more explicit control the `add_column()` and `add_columns()` methods can be used to add one or multiple columns to a table. In both cases the new columns must be specified as `Column` or `MaskedColumn` objects with the name defined:

```
>>> from astropy.table import Column
>>> aa = Column(np.arange(5), name='aa')
>>> t.add_column(aa, index=0) # Insert before the first table column

# Make a new table with the same number of rows and add columns to original table
>>> t2 = Table(np.arange(25).reshape(5, 5), names=('e', 'f', 'g', 'h', 'i'))
>>> t.add_columns(t2.columns.values())
```

Finally, columns can also be added from `Quantity` objects, which automatically sets the `.unit` attribute on the column:

```
>>> from astropy import units as u
>>> t['d'] = np.arange(1., 6.) * u.m
>>> t['d']
<Column name='d' unit='m' format=None description=None>
array([ 1.,  2.,  3.,  4.,  5.]
```

Remove columns

```
>>> t.remove_column('f')
>>> t.remove_columns(['aa', 'd1', 'd2', 'd3', 'e'])
>>> del t['g']
>>> del t['h', 'i']
>>> t.keep_columns(['a', 'b'])
```

Rename columns

```
>>> t.rename_column('a', 'a_new')
>>> t['b'].name = 'b_new'
```

Add a row of data

```
>>> t.add_row([-8, -9])
```

Remove rows

```
>>> t.remove_row(0)
>>> t.remove_rows(slice(4, 5))
>>> t.remove_rows([1, 2])
```

Sort by one more more columns

```
>>> t.sort('b_new')
>>> t.sort(['a_new', 'b_new'])
```

Reverse table rows

```
>>> t.reverse()
```

Modify meta-data

```
>>> t.meta['key'] = 'value'
```

Select or reorder columns

A new table with a subset or reordered list of columns can be created as shown in the following example:

```
>>> t = Table(arr, names=('a', 'b', 'c'))
>>> t_acb = t['a', 'c', 'b']
```

Another way to do the same thing is to provide a list or tuple as the item as shown below:

```
>>> new_order = ['a', 'c', 'b'] # List or tuple
>>> t_acb = t[new_order]
```

Caveats

Modifying the table data and properties is fairly straightforward. There are only a few things to keep in mind:

- The data type for a column cannot be changed in place. In order to do this you must make a copy of the table with the column type changed appropriately.
- Adding or removing a column will generate a new copy in memory of all the data. If the table is very large this may be slow.
- Adding a row *may* require a new copy in memory of the table data. This depends on the detailed layout of Python objects in memory and cannot be reliably controlled. In some cases it may be possible to build a table row by row in less than $O(N^2)$ time but you cannot count on it.

Another subtlety to keep in mind are cases where the return value of an operation results in a new table in memory versus a view of the existing table data. As an example, imagine trying to set two table elements using column selection with `t['a', 'c']` in combination with row index selection:

```
>>> t = Table([[1, 2], [3, 4], [5, 6]], names=('a', 'b', 'c'))
>>> t['a', 'c'][1] = (100, 100)
>>> print t
  a   b   c
---  ---  ---
  1   3   5
  2   4   6
```

This might be surprising because the data values did not change and there was no error. In fact what happened is that `t['a', 'c']` created a new temporary table in memory as a *copy* of the original and then updated row 1 of the copy. The original `t` table was unaffected and the new temporary table disappeared once the statement was complete. The takeaway is to pay attention to how certain operations are performed one step at a time.

8.3.4 Table operations

Table operations

In this section we describe higher-level operations that can be used to generate a new table from one or more input tables. This includes:

Documentation	Description	Function
Grouped operations	Group tables and columns by keys	<code>group_by</code>
Stack vertically	Concatenate input tables along rows	<code>vstack</code>
Stack horizontally	Concatenate input tables along columns	<code>hstack</code>
Join	Database-style join of two tables	<code>join</code>

Grouped operations

Sometimes in a table or table column there are natural groups within the dataset for which it makes sense to compute some derived values. A simple example is a list of objects with photometry from various observing runs:

```
>>> from astropy.table import Table
>>> obs = Table.read("""name      obs_date      mag_b      mag_v
...                M31          2012-01-02  17.0      17.5
...                M31          2012-01-02  17.1      17.4
...                M101         2012-01-02  15.1      13.5
...                M82          2012-02-14  16.2      14.5
...                M31          2012-02-14  16.9      17.3
...                M82          2012-02-14  15.2      15.5
...                M101         2012-02-14  15.0      13.6
...                M82          2012-03-26  15.7      16.5
...                M101         2012-03-26  15.1      13.5
...                M101         2012-03-26  14.8      14.3
...                """, format='ascii')
```

Table groups Now suppose we want the mean magnitudes for each object. We first group the data by the name column with the `group_by()` method. This returns a new table sorted by name which has a `groups` property specifying the unique values of name and the corresponding table rows:

```
>>> obs_by_name = obs.group_by('name')
>>> print obs_by_name
name  obs_date  mag_b  mag_v
----  -
M101  2012-01-02  15.1  13.5  << First group (index=0, key='M101')
M101  2012-02-14  15.0  13.6
M101  2012-03-26  15.1  13.5
M101  2012-03-26  14.8  14.3
  M31  2012-01-02  17.0  17.5  << Second group (index=4, key='M31')
  M31  2012-01-02  17.1  17.4
  M31  2012-02-14  16.9  17.3
  M82  2012-02-14  16.2  14.5  << Third group (index=7, key='M83')
  M82  2012-02-14  15.2  15.5
  M82  2012-03-26  15.7  16.5
                                     << End of groups (index=10)
>>> print obs_by_name.groups.keys
name
----
M101
  M31
  M82
>>> print obs_by_name.groups.indices
[ 0  4  7 10]
```

The `groups` property is the portal to all grouped operations with tables and columns. It defines how the table is grouped via an array of the unique row key values and the indices of the group boundaries for those key values. The groups here correspond to the row slices 0:4, 4:7, and 7:10 in the `obs_by_name` table.

The initial argument (`keys`) for the `group_by` function can take a number of input data types:

- Single string value with a table column name (as shown above)
- List of string values with table column names
- Another `Table` or `Column` with same length as table

- Numpy structured array with same length as table
- Numpy homogeneous array with same length as table

In all cases the corresponding row elements are considered as a tuple of values which form a key value that is used to sort the original table and generate the required groups.

As an example, to get the average magnitudes for each object on each observing night, we would first group the table on both name and obs_date as follows:

```
>>> print obs.group_by(['name', 'obs_date']).groups.keys
name  obs_date
----  -
M101  2012-01-02
M101  2012-02-14
M101  2012-03-26
M31   2012-01-02
M31   2012-02-14
M82   2012-02-14
M82   2012-03-26
```

Manipulating groups Once you have applied grouping to a table then you can easily access the individual groups or subsets of groups. In all cases this returns a new grouped table. For instance to get the sub-table which corresponds to the second group (index=1) do:

```
>>> print obs_by_name.groups[1]
name  obs_date  mag_b  mag_v
----  -
M31   2012-01-02  17.0  17.5
M31   2012-01-02  17.1  17.4
M31   2012-02-14  16.9  17.3
```

To get the first and second groups together use a slice:

```
>>> groups01 = obs_by_name.groups[0:2]
>>> print groups01
name  obs_date  mag_b  mag_v
----  -
M101  2012-01-02  15.1  13.5
M101  2012-02-14  15.0  13.6
M101  2012-03-26  15.1  13.5
M101  2012-03-26  14.8  14.3
M31   2012-01-02  17.0  17.5
M31   2012-01-02  17.1  17.4
M31   2012-02-14  16.9  17.3
>>> print groups01.groups.keys
name
----
M101
M31
```

You can also supply a numpy array of indices or a boolean mask to select particular groups, e.g.:

```
>>> mask = obs_by_name.groups.keys['name'] == 'M101'
>>> print obs_by_name.groups[mask]
name  obs_date  mag_b  mag_v
----  -
M101  2012-01-02  15.1  13.5
M101  2012-02-14  15.0  13.6
```

```
M101 2012-03-26 15.1 13.5
M101 2012-03-26 14.8 14.3
```

One can iterate over the group sub-tables and corresponding keys with:

```
>>> from itertools import izip
>>> for key, group in izip(obs_by_name.groups.keys, obs_by_name.groups):
...     print('***** {0} *****'.format(key['name']))
...     print group
...     print
...
***** M101 *****
name  obs_date  mag_b mag_v
----  -
M101  2012-01-02  15.1  13.5
M101  2012-02-14  15.0  13.6
M101  2012-03-26  15.1  13.5
M101  2012-03-26  14.8  14.3
***** M31 *****
name  obs_date  mag_b mag_v
----  -
M31   2012-01-02  17.0  17.5
M31   2012-01-02  17.1  17.4
M31   2012-02-14  16.9  17.3
***** M82 *****
name  obs_date  mag_b mag_v
----  -
M82   2012-02-14  16.2  14.5
M82   2012-02-14  15.2  15.5
M82   2012-03-26  15.7  16.5
```

Column Groups Like `Table` objects, `Column` objects can also be grouped for subsequent manipulation with grouped operations. This can apply both to columns within a `Table` or bare `Column` objects.

As for `Table`, the grouping is generated with the `group_by` method. The difference here is that there is no option of providing one or more column names since that doesn't make sense for a `Column`.

Examples:

```
>>> from astropy.table import Column
>>> import numpy as np
>>> c = Column([1, 2, 3, 4, 5, 6], name='a')
>>> key_vals = np.array(['foo', 'bar', 'foo', 'foo', 'qux', 'qux'])
>>> cg = c.group_by(key_vals)

>>> for key, group in izip(cg.groups.keys, cg.groups):
...     print('***** {0} *****'.format(key))
...     print group
...     print
...
***** bar *****
a
---
2
***** foo *****
a
---
1
```

```

3
4
***** qux *****
a
---
5
6

```

Aggregation Aggregation is the process of applying a specified reduction function to the values within each group for each non-key column. This function must accept a numpy array as the first argument and return a single scalar value. Common function examples are `numpy.sum`, `numpy.mean`, and `numpy.std`.

For the example grouped table `obs_by_name` from above we compute the group means with the `aggregate` method:

```

>>> obs_mean = obs_by_name.groups.aggregate(np.mean)
WARNING: Cannot aggregate column 'obs_date' [astropy.table.groups]
>>> print obs_mean
name mag_b mag_v
---- -
M101 15.0 13.725
M31 17.0 17.4
M82 15.7 15.5

```

It seems the magnitude values were successfully averaged, but what about the WARNING? Since the `obs_date` column is a string-type array, the `numpy.mean` function failed and raised an exception. Any time this happens then `aggregate` will issue a warning and then drop that column from the output result. Note that the `name` column is one of the keys used to determine the grouping so it is automatically ignored from aggregation.

From a grouped table it is possible to select one or more columns on which to perform the aggregation:

```

>>> print obs_by_name['mag_b'].groups.aggregate(np.mean)
mag_b
----
15.0
17.0
15.7

>>> print obs_by_name['name', 'mag_v', 'mag_b'].groups.aggregate(np.mean)
name mag_v mag_b
---- -
M101 13.725 15.0
M31 17.4 17.0
M82 15.5 15.7

```

A single column of data can be aggregated as well:

```

>>> c = Column([1, 2, 3, 4, 5, 6], name='a')
>>> key_vals = np.array(['foo', 'bar', 'foo', 'foo', 'qux', 'qux'])
>>> cg = c.group_by(key_vals)
>>> cg_sums = cg.groups.aggregate(np.sum)
>>> for key, cg_sum in izip(cg.groups.keys, cg_sums):
...     print 'Sum for {0} = {1}'.format(key, cg_sum)
...
Sum for bar = 2
Sum for foo = 8
Sum for qux = 11

```

If the specified function has a `numpy.ufunc.reduceat` method, this will be called instead. This can improve the performance by a factor of 10 to 100 (or more) for large unmasked tables or columns with many relatively small groups. It also allows for the use of certain numpy functions which normally take more than one input array but also work as reduction functions, like `numpy.add`. The numpy functions which should take advantage of using `numpy.ufunc.reduceat` include:

```
numpy.add, numpy.arctan2, numpy.bitwise_and, numpy.bitwise_or, numpy.bitwise_xor,
numpy.copysign, numpy.divide, numpy.equal, numpy.floor_divide, numpy.fmax,
numpy.fmin, numpy.fmod, numpy.greater_equal, numpy.greater, numpy.hypot,
numpy.left_shift, numpy.less_equal, numpy.less, numpy.logaddexp2, numpy.logaddexp,
numpy.logical_and, numpy.logical_or, numpy.logical_xor, numpy.maximum,
numpy.minimum, numpy.mod, numpy.multiply, numpy.not_equal, numpy.power,
numpy.remainder, numpy.right_shift, numpy.subtract and numpy.true_divide.
```

As special cases `numpy.sum` and `numpy.mean` are substituted with their respective `reduceat` methods.

Filtering Table groups can be filtered by means of the `filter` method. This is done by supplying a function which is called for each group. The function which is passed to this method must accept two arguments:

- `table`: Table object
- `key_colnames`: list of columns in `table` used as keys for grouping

It must then return either `True` or `False`. As an example, the following will select all table groups with only positive values in the non-key columns:

```
>>> def all_positive(table, key_colnames):
...     colnames = [name for name in table.colnames if name not in key_colnames]
...     for colname in colnames:
...         if np.any(table[colname] < 0):
...             return False
...     return True
```

An example of using this function is:

```
>>> t = Table.read(""" a  b  c
...                   -2  7.0  0
...                   -2  5.0  1
...                   1  3.0  -5
...                   1 -2.0  -6
...                   1  1.0  7
...                   0  0.0  4
...                   3  3.0  5
...                   3 -2.0  6
...                   3  1.0  7""", format='ascii')
>>> tg = t.group_by('a')
>>> t_positive = tg.groups.filter(all_positive)
>>> for group in t_positive.groups:
...     print group
...     print
...
 a  b  c
--- --- ---
-2 7.0  0
-2 5.0  1

 a  b  c
--- --- ---
 0 0.0  4
```

As can be seen only the groups with `a == -2` and `a == 0` have all positive values in the non-key columns, so those are the ones that are selected.

Likewise a grouped column can be filtered with the `filter`, method but in this case the filtering function takes only a single argument which is the column group. It still must return either `True` or `False`. For example:

```
def all_positive(column):
    if np.any(column < 0):
        return False
    return True
```

Stack vertically

The `Table` class supports stacking tables vertically with the `vstack` function. This process is also commonly known as concatenating or appending tables in the row direction. It corresponds roughly to the `numpy.vstack` function.

For example, suppose one has two tables of observations with several column names in common:

```
>>> from astropy.table import Table, vstack
>>> obs1 = Table.read("""name      obs_date   mag_b   logLx
...                          M31      2012-01-02  17.0   42.5
...                          M82      2012-10-29  16.2   43.5
...                          M101     2012-10-31  15.1   44.5""", format='ascii')
```

```
>>> obs2 = Table.read("""name      obs_date   logLx
...                          NGC3516  2011-11-11  42.1
...                          M31      1999-01-05  43.1
...                          M82      2012-10-30  45.0""", format='ascii')
```

Now we can stack these two tables:

```
>>> print vstack([obs1, obs2])
name      obs_date   mag_b   logLx
-----
M31 2012-01-02  17.0   42.5
M82 2012-10-29  16.2   43.5
M101 2012-10-31  15.1   44.5
NGC3516 2011-11-11  --     42.1
M31 1999-01-05  --     43.1
M82 2012-10-30  --     45.0
```

Notice that the `obs2` table is missing the `mag_b` column, so in the stacked output table those values are marked as missing. This is the default behavior and corresponds to `join_type='outer'`. There are two other allowed values for the `join_type` argument, `'inner'` and `'exact'`:

```
>>> print vstack([obs1, obs2], join_type='inner')
name      obs_date   logLx
-----
M31 2012-01-02  42.5
M82 2012-10-29  43.5
M101 2012-10-31  44.5
NGC3516 2011-11-11  42.1
M31 1999-01-05  43.1
M82 2012-10-30  45.0
```

```
>>> print vstack([obs1, obs2], join_type='exact')
Traceback (most recent call last):
...
```

```
TableMergeError: Inconsistent columns in input arrays (use 'inner'
or 'outer' join_type to allow non-matching columns)
Traceback (most recent call last):
```

```
...
TableMergeError: Inconsistent columns in input arrays (use 'inner'
```

In the case of `join_type='inner'`, only the common columns (the intersection) are present in the output table. When `join_type='exact'` is specified then `vstack` requires that all the input tables have exactly the same column names.

More than two tables can be stacked by supplying a list of table objects:

```
>>> obs3 = Table.read("""name      obs_date      mag_b      logLx
...                          M45      2012-02-03  15.0      40.5""", format='ascii')
>>> print vstack([obs1, obs2, obs3])
name      obs_date      mag_b      logLx
-----
M31 2012-01-02  17.0      42.5
M82 2012-10-29  16.2      43.5
M101 2012-10-31  15.1      44.5
NGC3516 2011-11-11    --      42.1
M31 1999-01-05    --      43.1
M82 2012-10-30    --      45.0
M45 2012-02-03  15.0      40.5
```

See also the sections on [Merging metadata](#) and [Merging column attributes](#) for details on how these characteristics of the input tables are merged in the single output table. Note also that you can use a single table row instead of a full table as one of the inputs.

Stack horizontally

The `Table` class supports stacking tables horizontally (in the column-wise direction) with the `hstack` function. It corresponds roughly to the `numpy.hstack` function.

For example, suppose one has the following two tables:

```
>>> from astropy.table import Table, hstack
>>> t1 = Table.read("""a      b      c
...                    1      foo  1.4
...                    2      bar  2.1
...                    3      baz  2.8""", format='ascii')
>>> t2 = Table.read("""d      e
...                    ham  eggs
...                    spam toast""", format='ascii')
```

Now we can stack these two tables horizontally:

```
>>> print hstack([t1, t2])
a      b      c      d      e
----
1 foo 1.4  ham  eggs
2 bar 2.1  spam toast
3 baz 2.8  --    --
```

As with `vstack`, there is an optional `join_type` argument that can take values `'inner'`, `'exact'`, and `'outer'`. The default is `'outer'`, which effectively takes the union of available rows and masks out any missing values. This is illustrated in the example above. The other options give the intersection of rows, where `'exact'` requires that all tables have exactly the same number of rows:

```
>>> print hstack([t1, t2], join_type='inner')
  a  b  c  d  e
---  ---  ---  ---  ---
  1 foo 1.4 ham eggs
  2 bar 2.1 spam toast

>>> print hstack([t1, t2], join_type='exact')
Traceback (most recent call last):
...
TableMergeError: Inconsistent number of rows in input arrays (use 'inner' or
'outer' join_type to allow non-matching rows)
Traceback (most recent call last):
...
TableMergeError: Inconsistent number of rows in input arrays (use 'inner' or
```

More than two tables can be stacked by supplying a list of table objects. The example below also illustrates the behavior when there is a conflict in the input column names (see the section on [Column renaming](#) for details):

```
>>> t3 = Table.read("""a  b
...                    M45  2012-02-03""", format='ascii')
>>> print hstack([t1, t2, t3])
a_1 b_1 c  d  e  a_3  b_3
---  ---  ---  ---  ---  ---  ---
  1 foo 1.4 ham eggs M45 2012-02-03
  2 bar 2.1 spam toast --  --
  3 baz 2.8 -- -- --  --
```

The metadata from the input tables is merged by the process described in the [Merging metadata](#) section. Note also that you can use a single table row instead of a full table as one of the inputs.

Join

The `Table` class supports the [database join](#) operation. This provides a flexible and powerful way to combine tables based on the values in one or more key columns.

For example, suppose one has two tables of observations, the first with B and V magnitudes and the second with X-ray luminosities of an overlapping (but not identical) sample:

```
>>> from astropy.table import Table, join
>>> optical = Table.read("""name  obs_date  mag_b  mag_v
...                          M31      2012-01-02  17.0  16.0
...                          M82      2012-10-29  16.2  15.2
...                          M101     2012-10-31  15.1  15.5""", format='ascii')
>>> xray = Table.read(""" name  obs_date  logLx
...                    NGC3516 2011-11-11 42.1
...                    M31     1999-01-05 43.1
...                    M82     2012-10-29 45.0""", format='ascii')
```

The `join()` method allows one to merge these two tables into a single table based on matching values in the “key columns”. By default the key columns are the set of columns that are common to both tables. In this case the key columns are `name` and `obs_date`. We can find all the observations of the same object on the same date as follows:

```
>>> opt_xray = join(optical, xray)
>>> print opt_xray
name  obs_date  mag_b  mag_v  logLx
---  ---  ---  ---  ---
  M82 2012-10-29  16.2  15.2  45.0
```

We can perform the match only by name by providing the `keys` argument, which can be either a single column name or a list of column names:

```
>>> print join(optical, xray, keys='name')
name obs_date_1 mag_b mag_v obs_date_2 logLx
-----
M31 2012-01-02 17.0 16.0 1999-01-05 43.1
M82 2012-10-29 16.2 15.2 2012-10-29 45.0
```

This output table has all observations that have both optical and X-ray data for an object (M31 and M82). Notice that since the `obs_date` column occurs in both tables it has been split into two columns, `obs_date_1` and `obs_date_2`. The values are taken from the “left” (optical) and “right” (xray) tables, respectively.

The table joins so far are known as “inner” joins and represent the strict intersection of the two tables on the key columns.

If one wants to make a new table which has *every* row from the left table and includes matching values from the right table when available, this is known as a left join:

```
>>> print join(optical, xray, join_type='left')
name obs_date mag_b mag_v logLx
-----
M101 2012-10-31 15.1 15.5 --
M31 2012-01-02 17.0 16.0 --
M82 2012-10-29 16.2 15.2 45.0
```

Two of the observations do not have X-ray data, as indicated by the “-” in the table. When there are any missing values the output will be a masked table. You might be surprised that there is no X-ray data for M31 in the output. Remember that the default matching key includes both name and `obs_date`. Specifying the key as only the name column gives:

```
>>> print join(optical, xray, join_type='left', keys='name')
name obs_date_1 mag_b mag_v obs_date_2 logLx
-----
M101 2012-10-31 15.1 15.5 -- --
M31 2012-01-02 17.0 16.0 1999-01-05 43.1
M82 2012-10-29 16.2 15.2 2012-10-29 45.0
```

Likewise one can construct a new table with every row of the right table and matching left values (when available) using `join_type='right'`.

Finally, to make a table with the union of rows from both tables do an “outer” join:

```
>>> print join(optical, xray, join_type='outer')
name obs_date mag_b mag_v logLx
-----
M101 2012-10-31 15.1 15.5 --
M31 1999-01-05 -- -- 43.1
M31 2012-01-02 17.0 16.0 --
M82 2012-10-29 16.2 15.2 45.0
NGC3516 2011-11-11 -- -- 42.1
```

Identical keys The `Table` join operation works even if there are multiple rows with identical key values. For example the following tables have multiple rows for the key column `x`:

```
>>> from astropy.table import Table, join
>>> left = Table([[0, 1, 1, 2], ['L1', 'L2', 'L3', 'L4']], names=('key', 'L'))
>>> right = Table([[1, 1, 2, 4], ['R1', 'R2', 'R3', 'R4']], names=('key', 'R'))
>>> print left
```

```

key  L
----
  0  L1
  1  L2
  1  L3
  2  L4
>>> print right
key  R
----
  1  R1
  1  R2
  2  R3
  4  R4

```

Doing an outer join on these tables shows that what is really happening is a [Cartesian product](#). For each matching key, every combination of the left and right tables is represented. When there is no match in either the left or right table, the corresponding column values are designated as missing.

```

win32

>>> print join(left, right, join_type='outer')
key  L  R
----
  0  L1  --
  1  L2  R1
  1  L2  R2
  1  L3  R1
  1  L3  R2
  2  L4  R3
  4  --  R4

```

Note: The output table is sorted on the key columns, but when there are rows with identical keys the output order in the non-key columns is not guaranteed to be identical across installations. In the example above the order within the four rows with `key == 1` can vary.

An inner join is the same but only returns rows where there is a key match in both the left and right tables:

```

win32

>>> print join(left, right, join_type='inner')
key  L  R
----
  1  L2  R1
  1  L2  R2
  1  L3  R1
  1  L3  R2
  2  L4  R3

```

Conflicts in the input table names are handled by the process described in the section on [Column renaming](#). See also the sections on [Merging metadata](#) and [Merging column attributes](#) for details on how these characteristics of the input tables are merged in the single output table.

Merging details

When combining two or more tables there is the need to merge certain characteristics in the inputs and potentially resolve conflicts. This section describes the process.

Column renaming In cases where the input tables have conflicting column names, there is a mechanism to generate unique output column names. There are two keyword arguments that control the renaming behavior:

table_names

Two-element list of strings that provide a name for the tables being joined. By default this is ['1', '2', ...], where the numbers correspond to the input tables.

uniq_col_name

String format specifier with a default value of '{col_name}_{table_name}'.

This is most easily understood by example using the `optical` and `xray` tables in the `join()` example defined previously:

```
>>> print join(optical, xray, keys='name',
...            table_names=['OPTICAL', 'XRAY'],
...            uniq_col_name='{table_name}_{col_name}')
name OPTICAL_obs_date mag_b mag_v XRAY_obs_date logLx
-----
M31      2012-01-02  17.0  16.0    1999-01-05  43.1
M82      2012-10-29  16.2  15.2    2012-10-29  45.0
```

Merging metadata `Table` objects can have associated metadata:

- `Table.meta`: table-level metadata as an ordered dictionary
- `Column.meta`: per-column metadata as an ordered dictionary

The table operations described here handle the task of merging the metadata in the input tables into a single output structure. Because the metadata can be arbitrarily complex there is no unique way to do the merge. The current implementation uses a simple recursive algorithm with four rules:

- `dict` elements are merged by keys
- Conflicting `list` or `tuple` elements are concatenated
- Conflicting `dict` elements are merged by recursively calling the merge function
- **Conflicting elements that are not both `list`, `tuple`, or `dict` will follow the following rules:**
 - If both metadata values are identical, the output is set to this value
 - If one of the conflicting metadata values is `None`, the other value is picked
 - If both metadata values are different and neither is `None`, the one for the last table in the list is picked

By default, a warning is emitted in the last case (both metadata values are not `None`). The warning can be silenced or made into an exception using the `metadata_conflicts` argument to `hstack()`, `vstack()`, or `join()`. The `metadata_conflicts` option can be set to:

- `'silent'` - no warning is emitted, the value for the last table is silently picked
- `'warn'` - a warning is emitted, the value for the last table is picked
- `'error'` - an exception is raised

Merging column attributes In addition to the table and column `meta` attributes, the column attributes `unit`, `format`, and `description` are merged by going through the input tables in order and taking the first value which is defined (i.e. is not `None`). For example:

```

>>> from astropy.table import Column, Table, vstack
>>> col1 = Column([1], name='a')
>>> col2 = Column([2], name='a', unit='cm')
>>> col3 = Column([3], name='a', unit='m')
>>> t1 = Table([col1])
>>> t2 = Table([col2])
>>> t3 = Table([col3])
>>> out = vstack([t1, t2, t3])
WARNING: MergeConflictWarning: In merged column 'a' the 'unit' attribute does
not match (cm != m). Using m for merged output [astropy.table.operations]
>>> out['a'].unit
Unit("m")

```

The rules for merging are as for [Merging metadata](#), and the `metadata_conflicts` option also controls the merging of column attributes.

8.3.5 Masking

Masking and missing values

The `astropy.table` package provides support for masking and missing values in a table by wrapping the `numpy.ma` masked array package. This allows handling tables with missing or invalid entries in much the same manner as for standard (unmasked) tables. It is useful to be familiar with the [masked array](#) documentation when using masked tables within `astropy.table`.

In a nutshell, the concept is to define a boolean mask that mirrors the structure of the table data array. Wherever a mask value is `True`, the corresponding entry is considered to be missing or invalid. Operations involving column or row access and slicing are unchanged. The key difference is that arithmetic or reduction operations involving columns or column slices follow the rules for [operations on masked arrays](#).

Note: Reduction operations like `numpy.sum` or `numpy.mean` follow the convention of ignoring masked (invalid) values. This differs from the behavior of the floating point NaN, for which the sum of an array including one or more NaN's will result in NaN. See <http://numpy.scipy.org/NA-overview.html> for a very interesting discussion of different strategies for handling missing data in the context of `numpy`.

Table creation

A masked table can be created in several ways:

Create a new table object and specify `masked=True`

```

>>> from astropy.table import Table, Column, MaskedColumn
>>> t = Table([(1, 2), (3, 4)], names=('a', 'b'), masked=True)
>>> t
<Table rows=2 names=('a', 'b')>
masked_array(data = [(1, 3) (2, 4)],
             mask = [(False, False) (False, False)],
             fill_value = (999999, 999999),
             dtype = [('a', '<i8'), ('b', '<i8')])

```

Notice the table attributes `mask` and `fill_value` that are available for a masked table.

Create a table with one or more columns as a `MaskedColumn` object

```
>>> a = MaskedColumn([1, 2], name='a')
>>> b = Column([3, 4], name='b')
>>> t = Table([a, b])
```

The `MaskedColumn` is the masked analog of the `Column` class and provides the interface for creating and manipulating a column of masked data. The `MaskedColumn` class inherits from `numpy.ma.MaskedArray`, in contrast to `Column` which inherits from `numpy.ndarray`. This distinction is the main reason there are different classes for these two cases.

Create a table with one or more columns as a numpy MaskedArray

```
>>> from numpy import ma # masked array package
>>> a = ma.array([1, 2])
>>> b = [3, 4]
>>> t = Table([a, b], names=('a', 'b'))
```

Add a MaskedColumn object to an existing table

```
>>> t = Table([[1, 2]], names=['a'])
>>> b = MaskedColumn([3, 4], mask=[True, False])
>>> t['b'] = b
```

INFO: Upgrading Table to masked Table. Use `Table.filled()` to convert to unmasked table. [astropy.table]

Note the INFO message because the underlying type of the table is modified in this operation.

Add a new row to an existing table and specify a mask argument

```
>>> a = Column([1, 2], name='a')
>>> b = Column([3, 4], name='b')
>>> t = Table([a, b])
>>> t.add_row([3, 6], mask=[True, False])
```

INFO: Upgrading Table to masked Table. Use `Table.filled()` to convert to unmasked table. [astropy.table]

Convert an existing table to a masked table

```
>>> t = Table([[1, 2], ['x', 'y']]) # standard (unmasked) table
>>> t = Table(t, masked=True) # convert to masked table
```

Table access

Nearly all the of standard methods for accessing and modifying data columns, rows, and individual elements also apply to masked tables.

There are two minor differences for the `Row` object that is obtained by indexing a single row of a table:

- For standard tables, two such rows can be compared for equality, but in masked tables this comparison will produce an exception.
- For standard tables a `Row` object provides a view of the underlying table data so that it is possible to modify a table by modifying the row values. In masked tables this is a copy so that modifying the `Row` object has no effect on the original table data.

Both of these differences are due to issues in the underlying `numpy.ma.MaskedArray` implementation.

Masking and filling

Both the `Table` and `MaskedColumn` classes provide attributes and methods to support manipulating tables with missing or invalid data.

Mask The actual mask for the table as a whole or a single column can be viewed and modified via the `mask` attribute:

```
>>> t = Table([(1, 2), (3, 4)], names=('a', 'b'), masked=True)
>>> t['a'].mask = [False, True] # Modify column mask (boolean array)
>>> t['b'].mask = [True, False] # Modify column mask (boolean array)
>>> print(t)
  a   b
--- ---
  1  --
--   4
```

Masked entries are shown as `--` when the table is printed.

Filling The entries which are masked (i.e. missing or invalid) can be replaced with specified fill values. In this case the `MaskedColumn` or masked `Table` will be converted to a standard `Column` or table. Each column in a masked table has a `fill_value` attribute that specifies the default fill value for that column. To perform the actual replacement operation the `filled()` method is called. This takes an optional argument which can override the default column `fill_value` attribute.

```
>>> t['a'].fill_value = -99
>>> t['b'].fill_value = 33

>>> print t.filled()
  a   b
--- ---
  1  33
-99   4

>>> print t['a'].filled()
  a
---
  1
-99

>>> print t['a'].filled(999)
  a
---
  1
999

>>> print t.filled(1000)
  a   b
---- ----
  1 1000
1000   4
```

8.3.6 I/O with tables

Reading and writing Table objects

Astropy provides a unified interface for reading and writing data in different formats. For many common cases this will simplify the process of file I/O and reduce the need to master the separate details of all the I/O packages within Astropy. For details and examples of using this interface see the *Unified file read/write interface* section.

Getting started

The `Table` class includes two methods, `read()` and `write()`, that make it possible to read from and write to files. A number of formats are automatically supported (see *Built-in table readers/writers*) and new file formats and extensions can be registered with the `Table` class (see *I/O Registry (astropy.io.registry)*).

To use this interface, first import the `Table` class, then simply call the `Table.read()` method with the name of the file and the file format, for instance `'ascii.daophot'`:

```
>>> from astropy.table import Table
>>> t = Table.read('photometry.dat', format='ascii.daophot')
```

It is possible to load tables directly from the Internet using URLs. For example, download tables from Vizier catalogues in CDS format (`'ascii.cds'`):

```
>>> t = Table.read("ftp://cdsarc.u-strasbg.fr/pub/cats/VII/253/snrs.dat",
...               readme="ftp://cdsarc.u-strasbg.fr/pub/cats/VII/253/ReadMe",
...               format="ascii.cds")
```

For certain file formats, the format can be automatically detected, for example from the filename extension:

```
>>> t = Table.read('table.tex')
```

Similarly, for writing, the format can be explicitly specified:

```
>>> t.write(filename, format='latex')
```

As for the `read()` method, the format may be automatically identified in some cases.

Any additional arguments specified will depend on the format. For examples of this see the section *Built-in table readers/writers*. This section also provides the full list of choices for the `format` argument.

Supported formats

The *Unified file read/write interface* has built-in support for the following data file formats:

- *ASCII formats*
- *HDF5*
- *FITS*
- *VO Tables*

8.4 Reference/API

8.4.1 astropy.table Module

Functions

<code>hstack(tables[, join_type, uniq_col_name, ...])</code>	Stack tables along columns (horizontally) A <code>join_type</code> of 'exact' means that the
<code>join(left, right[, keys, join_type, ...])</code>	Perform a join of the left table with the right table on specified keys.
<code>vstack(tables[, join_type, metadata_conflicts])</code>	Stack tables vertically (along rows) A <code>join_type</code> of 'exact' means that the table

hstack

`astropy.table.hstack` (*tables*, *join_type*=u'outer', *uniq_col_name*=u'{col_name}_{table_name}', *table_names*=None, *metadata_conflicts*=u'warn')

Stack tables along columns (horizontally)

A *join_type* of 'exact' means that the tables must all have exactly the same number of rows. If *join_type* is 'inner' then the intersection of rows will be output. A value of 'outer' (default) means the output will have the union of all rows, with table values being masked where no common values are available.

Parameters

tables : List of Table objects

Tables to stack along columns (horizontally) with the current table

join_type : str

Join type ('inner' | 'exact' | 'outer'), default is 'outer'

uniq_col_name : str or None

String generate a unique output column name in case of a conflict. The default is '{col_name}_{table_name}'.

table_names : list of str or None

Two-element list of table names used when generating unique output column names. The default is ['1', '2', ..].

col_name_map : empty dict or None

If passed as a dict then it will be updated in-place with the mapping of output to input column names.

metadata_conflicts : str

How to proceed with metadata conflicts. This should be one of:

- 'silent': silently pick the last conflicting meta-data value
- 'warn': pick the last conflicting meta-data value, but emit a warning (default)
- 'error': raise an exception.

Examples

To stack two tables horizontally (along columns) do:

```
>>> from astropy.table import Table, hstack
>>> t1 = Table({'a': [1, 2], 'b': [3, 4]}, names=('a', 'b'))
>>> t2 = Table({'c': [5, 6], 'd': [7, 8]}, names=('c', 'd'))
>>> print(t1)
a  b
--- ---
 1  3
 2  4
>>> print(t2)
c  d
--- ---
 5  7
 6  8
```

```
>>> print(hstack([t1, t2]))
  a  b  c  d
---  ---  ---  ---
  1  3  5  7
  2  4  6  8
```

join

```
astropy.table.join(left, right, keys=None, join_type=u'inner', uniq_col_name=u'{col_name}_{table_name}',
                  table_names=[u'1', u'2'], metadata_conflicts=u'warn')
```

Perform a join of the left table with the right table on specified keys.

Parameters

left : Table object or a value that will initialize a Table object

Left side table in the join

right : Table object or a value that will initialize a Table object

Right side table in the join

keys : str or list of str

Name(s) of column(s) used to match rows of left and right tables. Default is to use all columns which are common to both tables.

join_type : str

Join type ('inner' | 'outer' | 'left' | 'right'), default is 'inner'

uniq_col_name : str or None

String generate a unique output column name in case of a conflict. The default is '{col_name}_{table_name}'.

table_names : list of str or None

Two-element list of table names used when generating unique output column names. The default is ['1', '2'].

metadata_conflicts : str

How to proceed with metadata conflicts. This should be one of:

- 'silent': silently pick the last conflicting meta-data value
- 'warn': pick the last conflicting meta-data value, but emit a warning (default)
- 'error': raise an exception.

vstack

```
astropy.table.vstack(tables, join_type=u'outer', metadata_conflicts=u'warn')
```

Stack tables vertically (along rows)

A `join_type` of 'exact' means that the tables must all have exactly the same column names (though the order can vary). If `join_type` is 'inner' then the intersection of common columns will be output. A value of 'outer' (default) means the output will have the union of all columns, with table values being masked where no common values are available.

Parameters**tables** : Table or list of Table objects

Table(s) to stack along rows (vertically) with the current table

join_type : str

Join type ('inner' | 'exact' | 'outer'), default is 'exact'

metadata_conflicts : str**How to proceed with metadata conflicts. This should be one of:**

- 'silent': silently pick the last conflicting meta-data value
- 'warn': pick the last conflicting meta-data value, but emit a warning (default)
- 'error': raise an exception.

Examples

To stack two tables along rows do:

```
>>> from astropy.table import vstack, Table
>>> t1 = Table({'a': [1, 2], 'b': [3, 4]}, names=('a', 'b'))
>>> t2 = Table({'a': [5, 6], 'b': [7, 8]}, names=('a', 'b'))
>>> print(t1)
a    b
---  ---
 1    3
 2    4
>>> print(t2)
a    b
---  ---
 5    7
 6    8
>>> print(vstack([t1, t2]))
a    b
---  ---
 1    3
 2    4
 5    7
 6    8
```

Classes

Column	Define a data column for use in a Table object.
ColumnGroups(parent_column[, indices, keys])	
Conf	Configuration parameters for <code>astropy.table</code> .
MaskedColumn	Define a masked data column for use in a Table object.
Row(table, index)	A class to represent one row of a Table object.
Table([data, masked, names, dtype, meta, ...])	A class to represent tables of heterogeneous data.
TableColumns([cols])	OrderedDict subclass for a set of columns.
TableFormatter	
TableGroups(parent_table[, indices, keys])	
TableMergeError	

Column

class `astropy.table.Column`

Bases: `astropy.table.column.BaseColumn`

Define a data column for use in a Table object.

Parameters

data : list, ndarray or None

Column data values

name : str

Column name and key for reference within Table

dtype : `numpy.dtype` compatible value

Data type for column

shape : tuple or ()

Dimensions of a single row element in the column data

length : int or 0

Number of row elements in column data

description : str or None

Full description of column

unit : str or None

Physical unit

format : str or None or function or callable

Format string for outputting column values. This can be an “old-style” (`format % value`) or “new-style” (`str.format`) format specification string or a function or any callable object that accepts a single value and returns a string.

meta : dict-like or None

Meta-data associated with the column

Examples

A Column can be created in two different ways:

- Provide a data value but not shape or length (which are inferred from the data).

Examples:

```
col = Column(data=[1, 2], name='name') # shape=(2,)
col = Column(data=[[1, 2], [3, 4]], name='name') # shape=(2, 2)
col = Column(data=[1, 2], name='name', dtype=float)
col = Column(data=np.array([1, 2]), name='name')
col = Column(data=['hello', 'world'], name='name')
```

The `dtype` argument can be any value which is an acceptable fixed-size data-type initializer for the `numpy.dtype()` method. See <http://docs.scipy.org/doc/numpy/reference/arrays.dtypes.html>. Examples include:

–Python non-string type (float, int, bool)

–Numpy non-string type (e.g. `np.float32`, `np.int64`, `np.bool`)

–Numpy.dtype array-protocol type strings (e.g. `'i4'`, `'f8'`, `'S15'`)

If no `dtype` value is provide then the type is inferred using `np.array(data)`.

- Provide length and optionally shape, but not data

Examples:

```
col = Column(name='name', length=5)
col = Column(name='name', dtype=int, length=10, shape=(3,4))
```

The default `dtype` is `np.float64`. The `shape` argument is the array shape of a single cell in the column.

Attributes Summary

`name`

Methods Summary

<code>convert_unit_to(new_unit[, equivalencies])</code>	Converts the values of the column in-place from the current unit to the given unit.
<code>copy([order, data, copy_data])</code>	Return a copy of the current instance.
<code>more([max_lines, show_name, show_unit])</code>	Interactively browse column with a paging interface.
<code>pformat([max_lines, show_name, show_unit])</code>	Return a list of formatted string representation of column values.
<code>pprint([max_lines, show_name, show_unit])</code>	Print a formatted string representation of column values.

Attributes Documentation

name

Methods Documentation

convert_unit_to (*new_unit, equivalencies=[]*)

Converts the values of the column in-place from the current unit to the given unit.

To change the unit associated with this column without actually changing the data values, simply set the `unit` property.

Parameters

new_unit : str or `astropy.units.UnitBase` instance

The unit to convert to.

equivalencies : list of equivalence pairs, optional

A list of equivalence pairs to try if the unit are not directly convertible. See *Equivalencies*.

Raises

astropy.units.UnitsError

If units are inconsistent

copy (*order='C', data=None, copy_data=True*)

Return a copy of the current instance.

If `data` is supplied then a view (reference) of `data` is used, and `copy_data` is ignored.

Parameters

order : {'C', 'F', 'A', 'K'}, optional

Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A' means 'F' if `a` is Fortran contiguous, 'C' otherwise. 'K' means match the layout of `a` as closely as possible. (Note that this function and `func:numpy.copy` are very similar, but have different default values for their `order=` arguments.) Default is 'C'.

data : array, optional

If supplied then use a view of `data` instead of the instance data. This allows copying the instance attributes and meta.

copy_data : bool, optional

Make a copy of the internal numpy array instead of using a reference. Default is True.

Returns

col: Column or MaskedColumn

Copy of the current column (same type as original)

more (*max_lines=None, show_name=True, show_unit=False*)

Interactively browse column with a paging interface.

Supported keys:

```
f, <space> : forward one page
b : back one page
r : refresh same page
n : next row
p : previous row
< : go to beginning
> : go to end
q : quit browsing
h : print this help
```

Parameters

max_lines : int

Maximum number of lines in table output

show_name : bool

Include a header row for column names (default=True)

show_unit : bool

Include a header row for unit (default=False)

pformat (*max_lines=None, show_name=True, show_unit=False*)

Return a list of formatted string representation of column values.

If no value of `max_lines` is supplied then the height of the screen terminal is used to set `max_lines`. If the terminal height cannot be determined then the default will be determined using the `astropy.conf.max_lines` configuration item. If a negative value of `max_lines` is supplied then there is no line limit applied.

Parameters**max_lines** : int

Maximum lines of output (header + data rows)

show_name : bool

Include column name (default=True)

show_unit : bool

Include a header row for unit (default=False)

Returns**lines** : list

List of lines with header and formatted column values

pprint (*max_lines=None, show_name=True, show_unit=False*)

Print a formatted string representation of column values.

If no value of `max_lines` is supplied then the height of the screen terminal is used to set `max_lines`. If the terminal height cannot be determined then the default will be determined using the `astropy.conf.max_lines` configuration item. If a negative value of `max_lines` is supplied then there is no line limit applied.

Parameters**max_lines** : int

Maximum number of values in output

show_name : bool

Include column name (default=True)

show_unit : bool

Include a header row for unit (default=False)

ColumnGroups**class** `astropy.table.ColumnGroups` (*parent_column, indices=None, keys=None*)Bases: `astropy.table.groups.BaseGroups`**Attributes Summary**

`indices``keys`

Methods Summary

`aggregate(func)``filter(func)` Filter groups in the Column based on evaluating function `func` on each group sub-table.

Attributes Documentation

indices

keys

Methods Documentation

aggregate (*func*)

filter (*func*)

Filter groups in the Column based on evaluating function *func* on each group sub-table.

The function which is passed to this method must accept one argument:

- *column* : Column object

It must then return either `True` or `False`. As an example, the following will select all column groups with only positive values:

```
def all_positive(column):  
    if np.any(column < 0):  
        return False  
    return True
```

Parameters

func : function

Filter function

Returns

out : Column

New column with the aggregated rows.

Conf

class `astropy.table.Conf`

Bases: `astropy.config.ConfigNamespace`

Configuration parameters for `astropy.table`.

Attributes Summary

<code>auto_colname</code>	The template that determines the name of a column if it cannot be determined.
---------------------------	---

Attributes Documentation

auto_colname

The template that determines the name of a column if it cannot be determined. Uses new-style (format method) string formatting.

MaskedColumn

class `astropy.table.MaskedColumn`

Bases: `astropy.table.Column`, `numpy.ma.core.MaskedArray`

Define a masked data column for use in a Table object.

Parameters

data : list, ndarray or None

Column data values

name : str

Column name and key for reference within Table

mask : list, ndarray or None

Boolean mask for which True indicates missing or invalid data

fill_value : float, int, str or None

Value used when filling masked column elements

dtype : `numpy.dtype` compatible value

Data type for column

shape : tuple or ()

Dimensions of a single row element in the column data

length : int or 0

Number of row elements in column data

description : str or None

Full description of column

unit : str or None

Physical unit

format : str or None or function or callable

Format string for outputting column values. This can be an “old-style” (`format % value`) or “new-style” (`str.format`) format specification string or a function or any callable object that accepts a single value and returns a string.

meta : dict-like or None

Meta-data associated with the column

Examples

A `MaskedColumn` is similar to a `Column` except that it includes `mask` and `fill_value` attributes. It can be created in two different ways:

- Provide a data value but not `shape` or `length` (which are inferred from the data).

Examples:

```
col = MaskedColumn(data=[1, 2], name='name')
col = MaskedColumn(data=[1, 2], name='name', mask=[True, False])
col = MaskedColumn(data=[1, 2], name='name', dtype=float, fill_value=99)
```

The `mask` argument will be cast as a boolean array and specifies which elements are considered to be missing or invalid.

The `dtype` argument can be any value which is an acceptable fixed-size data-type initializer for the `numpy.dtype()` method. See <http://docs.scipy.org/doc/numpy/reference/arrays.dtypes.html>. Examples include:

- Python non-string type (float, int, bool)
- Numpy non-string type (e.g. `np.float32`, `np.int64`, `np.bool`)
- Numpy.dtype array-protocol type strings (e.g. `'i4'`, `'f8'`, `'S15'`)

If no `dtype` value is provide then the type is inferred using `np.array(data)`. When `data` is provided then the `shape` and `length` arguments are ignored.

- Provide `length` and optionally `shape`, but not `data`

Examples:

```
col = MaskedColumn(name='name', length=5)
col = MaskedColumn(name='name', dtype=int, length=10, shape=(3,4))
```

The default `dtype` is `np.float64`. The `shape` argument is the array shape of a single cell in the column.

Attributes Summary

<code>data</code>
<code>fill_value</code>
<code>name</code>

Methods Summary

<code>convert_unit_to(new_unit[, equivalencies])</code>	Converts the values of the column in-place from the current unit to the given unit.
<code>copy([order, data, copy_data])</code>	Return a copy of the current instance.
<code>filled([fill_value])</code>	Return a copy of self, with masked values filled with a given value.
<code>more([max_lines, show_name, show_unit])</code>	Interactively browse column with a paging interface.
<code>pformat([max_lines, show_name, show_unit])</code>	Return a list of formatted string representation of column values.
<code>pprint([max_lines, show_name, show_unit])</code>	Print a formatted string representation of column values.

Attributes Documentation

data

fill_value

name

Methods Documentation

convert_unit_to (*new_unit*, *equivalencies=[]*)

Converts the values of the column in-place from the current unit to the given unit.

To change the unit associated with this column without actually changing the data values, simply set the `unit` property.

Parameters

new_unit : str or `astropy.units.UnitBase` instance

The unit to convert to.

equivalencies : list of equivalence pairs, optional

A list of equivalence pairs to try if the unit are not directly convertible. See *Equivalencies*.

Raises

`astropy.units.UnitsError`

If units are inconsistent

copy (*order='C'*, *data=None*, *copy_data=True*)

Return a copy of the current instance.

If `data` is supplied then a view (reference) of `data` is used, and `copy_data` is ignored.

Parameters

order : {'C', 'F', 'A', 'K'}, optional

Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A' means 'F' if `a` is Fortran contiguous, 'C' otherwise. 'K' means match the layout of `a` as closely as possible. (Note that this function and `func:numpy.copy` are very similar, but have different default values for their `order=` arguments.) Default is 'C'.

data : array, optional

If supplied then use a view of `data` instead of the instance `data`. This allows copying the instance attributes and meta.

copy_data : bool, optional

Make a copy of the internal numpy array instead of using a reference. Default is True.

Returns

col: Column or MaskedColumn

Copy of the current column (same type as original)

filled (*fill_value=None*)

Return a copy of self, with masked values filled with a given value.

Parameters

fill_value : scalar; optional

The value to use for invalid entries (`None` by default). If `None`, the `fill_value` attribute of the array is used instead.

Returns

filled_column : Column

A copy of `self` with masked entries replaced by `fill_value` (be it the function argument or the attribute of `self`).

more (*max_lines=None, show_name=True, show_unit=False*)

Interactively browse column with a paging interface.

Supported keys:

```
f, <space> : forward one page
b : back one page
r : refresh same page
n : next row
p : previous row
< : go to beginning
> : go to end
q : quit browsing
h : print this help
```

Parameters

max_lines : int

Maximum number of lines in table output

show_name : bool

Include a header row for column names (default=True)

show_unit : bool

Include a header row for unit (default=False)

pformat (*max_lines=None, show_name=True, show_unit=False*)

Return a list of formatted string representation of column values.

If no value of `max_lines` is supplied then the height of the screen terminal is used to set `max_lines`. If the terminal height cannot be determined then the default will be determined using the `astropy.conf.max_lines` configuration item. If a negative value of `max_lines` is supplied then there is no line limit applied.

Parameters

max_lines : int

Maximum lines of output (header + data rows)

show_name : bool

Include column name (default=True)

show_unit : bool

Include a header row for unit (default=False)

Returns

lines : list

List of lines with header and formatted column values

pprint (*max_lines=None, show_name=True, show_unit=False*)

Print a formatted string representation of column values.

If no value of `max_lines` is supplied then the height of the screen terminal is used to set `max_lines`. If the terminal height cannot be determined then the default will be determined using the `astropy.conf.max_lines` configuration item. If a negative value of `max_lines` is supplied then there is no line limit applied.

Parameters

max_lines : int

Maximum number of values in output

show_name : bool

Include column name (default=True)

show_unit : bool

Include a header row for unit (default=False)

Row

class `astropy.table.Row` (*table, index*)

Bases: `object`

A class to represent one row of a Table object.

A Row object is returned when a Table object is indexed with an integer or when iterating over a table:

```
>>> from astropy.table import Table
>>> table = Table([(1, 2), (3, 4)], names=('a', 'b'),
...              dtype=('int32', 'int32'))
>>> row = table[1]
>>> row
<Row 1 of table
  values=(2, 4)
  dtype=[('a', '<i4'), ('b', '<i4')]>
>>> row['a']
2
>>> row[1]
4
```

Attributes Summary

```
colnames
columns
data
dtype
index
meta
table
```

Attributes Documentation

colnames

columns

data

dtype

`index`

`meta`

`table`

Table

class `astropy.table.Table` (*data=None, masked=None, names=None, dtype=None, meta=None, copy=True, rows=None*)

Bases: `object`

A class to represent tables of heterogeneous data.

`Table` provides a class for heterogeneous tabular data, making use of a `numpy` structured array internally to store the data values. A key enhancement provided by the `Table` class is the ability to easily modify the structure of the table by adding or removing columns, or adding new rows of data. In addition table and column metadata are fully supported.

`Table` differs from `NDData` by the assumption that the input data consists of columns of homogeneous data, where each column has a unique identifier and may contain additional metadata such as the data unit, format, and description.

Parameters

data : `numpy ndarray`, `dict`, `list`, or `Table`, optional

Data to initialize table.

masked : `bool`, optional

Specify whether the table is masked.

names : `list`, optional

Specify column names

dtype : `list`, optional

Specify column data types

meta : `dict`, optional

Metadata associated with the table.

copy : `bool`, optional

Copy the input data (default=True).

rows : `numpy ndarray`, `list of lists`, optional

Row-oriented data for table instead of `data` argument

Attributes Summary

```
ColumnClass
colnames
dtype
```

Continued on next page

Table 8.11 – continued from previous page

```
groups
mask
masked
```

Methods Summary

<code>add_column(col[, index])</code>	Add a new Column object <code>col</code> to the table.
<code>add_columns(cols[, indexes])</code>	Add a list of new Column objects <code>cols</code> to the table.
<code>add_row([vals, mask])</code>	Add a new row to the end of the table.
<code>argsort([keys, kind])</code>	Return the indices which would sort the table according to one or more keys.
<code>convert_bytestring_to_unicode([python3_only])</code>	Convert bytestring columns (<code>dtype.kind='S'</code>) to unicode (<code>dtype.kind='U'</code>).
<code>convert_unicode_to_bytestring([python3_only])</code>	Convert ASCII-only unicode columns (<code>dtype.kind='U'</code>) to bytestring (<code>dtype.kind='S'</code>).
<code>copy([copy_data])</code>	Return a copy of the table.
<code>create_mask()</code>	
<code>field(item)</code>	Return <code>column[item]</code> for recarray compatibility.
<code>filled([fill_value])</code>	Return a copy of self, with masked values filled.
<code>group_by(keys)</code>	Group this table by the specified <code>keys</code> . This effectively splits the table into groups.
<code>index_column(name)</code>	Return the positional index of column <code>name</code> .
<code>keep_columns(names)</code>	Keep only the columns specified (remove the others).
<code>keys()</code>	
<code>more([max_lines, max_width, show_name, ...])</code>	Interactively browse table with a paging interface.
<code>next()</code>	Python 3 iterator
<code>pformat([max_lines, max_width, show_name, ...])</code>	Return a list of lines for the formatted string representation of the table.
<code>pprint([max_lines, max_width, show_name, ...])</code>	Print a formatted string representation of the table.
<code>read(*args, **kwargs)</code>	Read and parse a data table and return as a Table.
<code>remove_column(name)</code>	Remove a column from the table.
<code>remove_columns(names)</code>	Remove several columns from the table.
<code>remove_row(index)</code>	Remove a row from the table.
<code>remove_rows(row_specifier)</code>	Remove rows from the table.
<code>rename_column(name, new_name)</code>	Rename a column.
<code>reverse()</code>	Reverse the row order of table rows.
<code>show_in_browser([css, max_lines, jsviewer, ...])</code>	Render the table in HTML and show it in a web browser.
<code>sort(keys)</code>	Sort the table according to one or more keys.
<code>write(*args, **kwargs)</code>	Write this Table object out in the specified format.

Attributes Documentation**ColumnClass****colnames****dtype****groups****mask**

masked

Methods Documentation

add_column (*col*, *index=None*)

Add a new Column object *col* to the table. If *index* is supplied then insert column before *index* position in the list of columns, otherwise append column to the end of the list.

Parameters

col : Column

Column object to add.

index : int or None

Insert column before this position or at end (default)

Examples

Create a table with two columns 'a' and 'b':

```
>>> t = Table([[1, 2, 3], [0.1, 0.2, 0.3]], names=('a', 'b'))
>>> print(t)
 a  b
--- --
 1 0.1
 2 0.2
 3 0.3
```

Create a third column 'c' and append it to the end of the table:

```
>>> col_c = Column(name='c', data=['x', 'y', 'z'])
>>> t.add_column(col_c)
>>> print(t)
 a  b  c
--- -- ---
 1 0.1  x
 2 0.2  y
 3 0.3  z
```

Add column 'd' at position 1. Note that the column is inserted before the given index:

```
>>> col_d = Column(name='d', data=['a', 'b', 'c'])
>>> t.add_column(col_d, 1)
>>> print(t)
 a  d  b  c
--- -- ---
 1  a 0.1  x
 2  b 0.2  y
 3  c 0.3  z
```

To add several columns use `add_columns`.

add_columns (*cols*, *indexes=None*)

Add a list of new Column objects *cols* to the table. If a corresponding list of *indexes* is supplied then insert column before each *index* position in the *original* list of columns, otherwise append columns to the end of the list.

Parameters**cols** : list of Columns

Column objects to add.

indexes : list of ints or `None`

Insert column before this position or at end (default)

Examples

Create a table with two columns 'a' and 'b':

```
>>> t = Table([[1, 2, 3], [0.1, 0.2, 0.3]], names=('a', 'b'))
>>> print(t)
  a   b
---  ---
  1 0.1
  2 0.2
  3 0.3
```

Create column 'c' and 'd' and append them to the end of the table:

```
>>> col_c = Column(name='c', data=['x', 'y', 'z'])
>>> col_d = Column(name='d', data=['u', 'v', 'w'])
>>> t.add_columns([col_c, col_d])
>>> print(t)
  a   b   c   d
---  ---  ---  ---
  1 0.1   x   u
  2 0.2   y   v
  3 0.3   z   w
```

Add column 'c' at position 0 and column 'd' at position 1. Note that the columns are inserted before the given position:

```
>>> t = Table([[1, 2, 3], [0.1, 0.2, 0.3]], names=('a', 'b'))
>>> col_c = Column(name='c', data=['x', 'y', 'z'])
>>> col_d = Column(name='d', data=['u', 'v', 'w'])
>>> t.add_columns([col_c, col_d], [0, 1])
>>> print(t)
  c   a   d   b
---  ---  ---  ---
  x   1   u 0.1
  y   2   v 0.2
  z   3   w 0.3
```

add_row (*vals=None, mask=None*)

Add a new row to the end of the table.

The *vals* argument can be:**sequence (e.g. tuple or list)**

Column values in the same order as table columns.

mapping (e.g. dict)Keys corresponding to column names. Missing values will be filled with `np.zeros` for the column dtype.**None**All values filled with `np.zeros` for the column dtype.

This method requires that the Table object “owns” the underlying array data. In particular one cannot add a row to a Table that was initialized with `copy=False` from an existing array.

The `mask` attribute should give (if desired) the mask for the values. The type of the mask should match that of the values, i.e. if `vals` is an iterable, then `mask` should also be an iterable with the same length, and if `vals` is a mapping, then `mask` should be a dictionary.

Parameters

vals : tuple, list, dict or `None`

Use the specified values in the new row

mask : tuple, list, dict or `None`

Use the specified mask values in the new row

Examples

Create a table with three columns ‘a’, ‘b’ and ‘c’:

```
>>> t = Table([[1,2],[4,5],[7,8]], names=('a','b','c'))
>>> print(t)
  a  b  c
---  ---
  1  4  7
  2  5  8
```

Adding a new row with entries ‘3’ in ‘a’, ‘6’ in ‘b’ and ‘9’ in ‘c’:

```
>>> t.add_row([3,6,9])
>>> print(t)
  a  b  c
---  ---
  1  4  7
  2  5  8
  3  6  9
```

argsort (*keys=None, kind=None*)

Return the indices which would sort the table according to one or more key columns. This simply calls the `numpy.argsort` function on the table with the `order` parameter set to `keys`.

Parameters

keys : str or list of str

The column name(s) to order the table by

kind : {‘quicksort’, ‘mergesort’, ‘heapsort’}, optional

Sorting algorithm.

Returns

index_array : ndarray, int

Array of indices that sorts the table by the specified key column(s).

convert_bytestring_to_unicode (*python3_only=False*)

Convert bytestring columns (`dtype.kind='S'`) to unicode (`dtype.kind='U'`) assuming ASCII encoding.

Internally this changes string columns to represent each character in the string with a 4-byte UCS-4 equivalent, so it is inefficient for memory but allows Python 3 scripts to manipulate string arrays with natural syntax.

The `python3_only` parameter is provided as a convenience so that code can be written in a Python 2 / 3 compatible way:

```
>>> t = Table.read('my_data.fits')
>>> t.convert_bytestring_to_unicode (python3_only=True)
```

Parameters

python3_only : bool

Only do this operation for Python 3

convert_unicode_to_bytestring (*python3_only=False*)

Convert ASCII-only unicode columns (dtype.kind='U') to bytestring (dtype.kind='S').

When exporting a unicode string array to a file in Python 3, it may be desirable to encode unicode columns as bytestrings. This routine takes advantage of numpy automated conversion which works for strings that are pure ASCII.

The `python3_only` parameter is provided as a convenience so that code can be written in a Python 2 / 3 compatible way:

```
>>> t.convert_unicode_to_bytestring (python3_only=True)
>>> t.write('my_data.fits')
```

Parameters

python3_only : bool

Only do this operation for Python 3

copy (*copy_data=True*)

Return a copy of the table.

Parameters

copy_data : bool

If `True` (the default), copy the underlying data array. Otherwise, use the same data array

create_mask ()

field (*item*)

Return column[item] for recarray compatibility.

filled (*fill_value=None*)

Return a copy of self, with masked values filled.

If input `fill_value` supplied then that value is used for all masked entries in the table. Otherwise the individual `fill_value` defined for each table column is used.

Parameters

fill_value : str

If supplied, this `fill_value` is used for all masked entries in the entire table.

Returns

filled_table : Table

New table with masked values filled

group_by (*keys*)

Group this table by the specified *keys*

This effectively splits the table into groups which correspond to unique values of the *keys* grouping object. The output is a new `TableGroups` which contains a copy of this table but sorted by row according to *keys*.

The *keys* input to `group_by` can be specified in different ways:

- String or list of strings corresponding to table column name(s)
- Numpy array (homogeneous or structured) with same length as this table
- `Table` with same length as this table

Parameters

keys : str, list of str, numpy array, or `Table`

Key grouping object

Returns

out : `Table`

New table with groups set

index_column (*name*)

Return the positional index of column name.

Parameters

name : str

column name

Returns

index : int

Positional index of column name.

Examples

Create a table with three columns 'a', 'b' and 'c':

```
>>> t = Table([[1, 2, 3], [0.1, 0.2, 0.3], ['x', 'y', 'z']],
...           names=('a', 'b', 'c'))
>>> print(t)
  a   b   c
--- --- ---
  1 0.1  x
  2 0.2  y
  3 0.3  z
```

Get index of column 'b' of the table:

```
>>> t.index_column('b')
1
```

keep_columns (*names*)

Keep only the columns specified (remove the others).

Parameters

names : list

A list containing the names of the columns to keep. All other columns will be removed.

Examples

Create a table with three columns 'a', 'b' and 'c':

```
>>> t = Table([[1, 2, 3], [0.1, 0.2, 0.3], ['x', 'y', 'z']],
...           names=('a', 'b', 'c'))
>>> print(t)
  a   b   c
---  ---  ---
  1  0.1  x
  2  0.2  y
  3  0.3  z
```

Specifying only a single column name keeps only this column. Keep only column 'a' of the table:

```
>>> t.keep_columns('a')
>>> print(t)
  a
---
  1
  2
  3
```

Specifying a list of column names is keeps is also possible. Keep columns 'a' and 'c' of the table:

```
>>> t = Table([[1, 2, 3], [0.1, 0.2, 0.3], ['x', 'y', 'z']],
...           names=('a', 'b', 'c'))
>>> t.keep_columns(['a', 'c'])
>>> print(t)
  a   c
---  ---
  1   x
  2   y
  3   z
```

keys()

more (*max_lines=None, max_width=None, show_name=True, show_unit=None*)

Interactively browse table with a paging interface.

Supported keys:

```
f, <space> : forward one page
b : back one page
r : refresh same page
n : next row
p : previous row
< : go to beginning
> : go to end
q : quit browsing
h : print this help
```

Parameters

max_lines : int

Maximum number of lines in table output

max_width : int or `None`

Maximum character width of output

show_name : bool

Include a header row for column names (default=True)

show_unit : bool

Include a header row for unit. Default is to show a row for units only if one or more columns has a defined value for the unit.

next ()

Python 3 iterator

pformat (*max_lines=None, max_width=None, show_name=True, show_unit=None, html=False, tableid=None*)

Return a list of lines for the formatted string representation of the table.

If no value of `max_lines` is supplied then the height of the screen terminal is used to set `max_lines`. If the terminal height cannot be determined then the default is taken from the configuration item `astropy.conf.max_lines`. If a negative value of `max_lines` is supplied then there is no line limit applied.

The same applies for `max_width` except the configuration item is `astropy.conf.max_width`.

Parameters

max_lines : int or `None`

Maximum number of rows to output

max_width : int or `None`

Maximum character width of output

show_name : bool

Include a header row for column names (default=True)

show_unit : bool

Include a header row for unit. Default is to show a row for units only if one or more columns has a defined value for the unit.

html : bool

Format the output as an HTML table (default=False)

tableid : str or `None`

An ID tag for the table; only used if `html` is set. Default is “table{id}”, where `id` is the unique integer id of the table object, `id(self)`

Returns

lines : list

Formatted table as a list of strings

pprint (*max_lines=None, max_width=None, show_name=True, show_unit=None*)

Print a formatted string representation of the table.

If no value of `max_lines` is supplied then the height of the screen terminal is used to set `max_lines`. If the terminal height cannot be determined then the default is taken from the configuration item `astropy.conf.max_lines`. If a negative value of `max_lines` is supplied then there is no line limit applied.

The same applies for `max_width` except the configuration item is `astropy.conf.max_width`.

Parameters

max_lines : int

Maximum number of lines in table output

max_width : int or `None`

Maximum character width of output

show_name : bool

Include a header row for column names (default=True)

show_unit : bool

Include a header row for unit. Default is to show a row for units only if one or more columns has a defined value for the unit.

classmethod `read(*args, **kwargs)`

Read and parse a data table and return as a `Table`.

This function provides the `Table` interface to the astropy unified I/O layer. This allows easily reading a file in many supported data formats using syntax such as:

```
>>> from astropy.table import Table
>>> dat = Table.read('table.dat', format='ascii')
>>> events = Table.read('events.fits', format='fits')
```

The arguments and keywords (other than `format`) provided to this function are passed through to the underlying data reader (e.g. `read`).

The available built-in formats are:

Format	Read	Write	Auto-identify	Deprecated
ascii	Yes	Yes	No	
ascii.aastex	Yes	Yes	No	
ascii.basic	Yes	Yes	No	
ascii.cds	Yes	No	No	
ascii.commented_header	Yes	Yes	No	
ascii.csv	Yes	Yes	Yes	
ascii.daophot	Yes	No	No	
ascii.fixed_width	Yes	Yes	No	
ascii.fixed_width_no_header	Yes	Yes	No	
ascii.fixed_width_two_line	Yes	Yes	No	
ascii.html	Yes	Yes	Yes	
ascii.ipac	Yes	Yes	No	
ascii.latex	Yes	Yes	Yes	
ascii.no_header	Yes	Yes	No	
ascii.rdb	Yes	Yes	Yes	
ascii.sextractor	Yes	No	No	
ascii.tab	Yes	Yes	No	
fits	Yes	Yes	Yes	
hdf5	Yes	Yes	Yes	
votable	Yes	Yes	Yes	
aastex	Yes	Yes	No	Yes
cds	Yes	No	No	Yes
daophot	Yes	No	No	Yes
html	Yes	Yes	No	Yes
ipac	Yes	Yes	No	Yes
latex	Yes	Yes	No	Yes
rdb	Yes	Yes	No	Yes

Deprecated format names like `aastex` will be removed in a future version. Use the full name (e.g. `ascii.aastex`) instead.

remove_column (*name*)

Remove a column from the table.

This can also be done with:

```
del table[name]
```

Parameters

name : str

Name of column to remove

Examples

Create a table with three columns 'a', 'b' and 'c':

```
>>> t = Table([[1, 2, 3], [0.1, 0.2, 0.3], ['x', 'y', 'z']],
...           names=('a', 'b', 'c'))
>>> print(t)
a    b    c
---  ---  ---
  1  0.1  x
  2  0.2  y
  3  0.3  z
```

Remove column ‘b’ from the table:

```
>>> t.remove_column('b')
>>> print(t)
  a  c
---  ---
  1  x
  2  y
  3  z
```

To remove several columns at the same time use `remove_columns`.

remove_columns (*names*)

Remove several columns from the table.

Parameters

names : list

A list containing the names of the columns to remove

Examples

Create a table with three columns ‘a’, ‘b’ and ‘c’:

```
>>> t = Table([[1, 2, 3], [0.1, 0.2, 0.3], ['x', 'y', 'z']],
...           names=('a', 'b', 'c'))
>>> print(t)
  a  b  c
---  ---  ---
  1  0.1  x
  2  0.2  y
  3  0.3  z
```

Remove columns ‘b’ and ‘c’ from the table:

```
>>> t.remove_columns(['b', 'c'])
>>> print(t)
  a
---
  1
  2
  3
```

Specifying only a single column also works. Remove column ‘b’ from the table:

```
>>> t = Table([[1, 2, 3], [0.1, 0.2, 0.3], ['x', 'y', 'z']],
...           names=('a', 'b', 'c'))
>>> t.remove_columns('b')
>>> print(t)
  a  c
---  ---
  1  x
  2  y
  3  z
```

This gives the same as using `remove_column`.

remove_row (*index*)

Remove a row from the table.

Parameters**index** : int

Index of row to remove

Examples

Create a table with three columns 'a', 'b' and 'c':

```
>>> t = Table([[1, 2, 3], [0.1, 0.2, 0.3], ['x', 'y', 'z']],
...           names=('a', 'b', 'c'))
>>> print(t)
a   b   c
--- --- ---
 1 0.1  x
 2 0.2  y
 3 0.3  z
```

Remove row 1 from the table:

```
>>> t.remove_row(1)
>>> print(t)
a   b   c
--- --- ---
 1 0.1  x
 3 0.3  z
```

To remove several rows at the same time use `remove_rows`.**remove_rows** (*row_specifier*)

Remove rows from the table.

Parameters**row_specifier** : slice, int, or array of ints

Specification for rows to remove

Examples

Create a table with three columns 'a', 'b' and 'c':

```
>>> t = Table([[1, 2, 3], [0.1, 0.2, 0.3], ['x', 'y', 'z']],
...           names=('a', 'b', 'c'))
>>> print(t)
a   b   c
--- --- ---
 1 0.1  x
 2 0.2  y
 3 0.3  z
```

Remove rows 0 and 2 from the table:

```
>>> t.remove_rows([0, 2])
>>> print(t)
a   b   c
--- --- ---
 2 0.2  y
```

Note that there are no warnings if the slice operator extends outside the data:

```
>>> t = Table([[1, 2, 3], [0.1, 0.2, 0.3], ['x', 'y', 'z']],
...           names=('a', 'b', 'c'))
>>> t.remove_rows(slice(10, 20, 1))
>>> print(t)
  a   b   c
---  ---  ---
  1 0.1   x
  2 0.2   y
  3 0.3   z
```

rename_column (*name*, *new_name*)

Rename a column.

This can also be done directly with by setting the name attribute for a column:

```
table[name].name = new_name
```

Parameters

name : str

The current name of the column.

new_name : str

The new name for the column

Examples

Create a table with three columns ‘a’, ‘b’ and ‘c’:

```
>>> t = Table([[1,2],[3,4],[5,6]], names=('a', 'b', 'c'))
>>> print(t)
  a   b   c
---  ---  ---
  1   3   5
  2   4   6
```

Renaming column ‘a’ to ‘aa’:

```
>>> t.rename_column('a', 'aa')
>>> print(t)
 aa  b   c
---  ---  ---
  1   3   5
  2   4   6
```

reverse ()

Reverse the row order of table rows. The table is reversed in place and there are no function arguments.

Examples

Create a table with three columns:

```
>>> t = Table(['Max', 'Jo', 'John'], ['Miller', 'Miller', 'Jackson'],
...          [12,15,18]), names=('firstname', 'name', 'tel'))
>>> print(t)
firstname  name  tel
-----  -----  ---
```


The key(s) to order the table by

Examples

Create a table with 3 columns:

```
>>> t = Table([[ 'Max', 'Jo', 'John'], [ 'Miller', 'Miller', 'Jackson'],
...           [12, 15, 18]], names=('firstname', 'name', 'tel'))
>>> print(t)
-----
      Max  Miller  12
      Jo   Miller  15
      John Jackson 18
```

Sorting according to standard sorting rules, first 'name' then 'firstname':

```
>>> t.sort(['name', 'firstname'])
>>> print(t)
-----
      John Jackson 18
      Jo   Miller  15
      Max  Miller  12
```

write (*args, **kwargs)

Write this Table object out in the specified format.

This function provides the Table interface to the astropy unified I/O layer. This allows easily writing a file in many supported data formats using syntax such as:

```
>>> from astropy.table import Table
>>> dat = Table([[1, 2], [3, 4]], names=('a', 'b'))
>>> dat.write('table.dat', format='ascii')
```

The arguments and keywords (other than `format`) provided to this function are passed through to the underlying data reader (e.g. `write`).

The available built-in formats are:

Format	Read	Write	Auto-identify	Deprecated
ascii	Yes	Yes	No	
ascii.aastex	Yes	Yes	No	
ascii.basic	Yes	Yes	No	
ascii.commented_header	Yes	Yes	No	
ascii.csv	Yes	Yes	Yes	
ascii.fixed_width	Yes	Yes	No	
ascii.fixed_width_no_header	Yes	Yes	No	
ascii.fixed_width_two_line	Yes	Yes	No	
ascii.html	Yes	Yes	Yes	
ascii.ipac	Yes	Yes	No	
ascii.latex	Yes	Yes	Yes	
ascii.no_header	Yes	Yes	No	
ascii.rdb	Yes	Yes	Yes	
ascii.tab	Yes	Yes	No	
fits	Yes	Yes	Yes	
hdf5	Yes	Yes	Yes	
votable	Yes	Yes	Yes	
aastex	Yes	Yes	No	Yes
html	Yes	Yes	No	Yes
ipac	Yes	Yes	No	Yes
latex	Yes	Yes	No	Yes
rdb	Yes	Yes	No	Yes

Deprecated format names like `aastex` will be removed in a future version. Use the full name (e.g. `ascii.aastex`) instead.

TableColumns

class `astropy.table.TableColumns` (*cols={}*)

Bases: `collections.OrderedDict`

OrderedDict subclass for a set of columns.

This class enhances item access to provide convenient access to columns by name or index, including slice access. It also handles renaming of columns.

The initialization argument `cols` can be a list of `Column` objects or any structure that is valid for initializing a Python dict. This includes a dict, list of (key, val) tuples or [key, val] lists, etc.

Parameters

cols : dict, list, tuple; optional

Column objects as data structure that can init dict (see above)

Methods Summary

```
keys()
values()
```

Methods Documentation

keys ()

`values ()`

TableFormatter

```
class astropy.table.TableFormatter
    Bases: object
```

TableGroups

```
class astropy.table.TableGroups (parent_table, indices=None, keys=None)
    Bases: astropy.table.groups.BaseGroups
```

Attributes Summary

<code>indices</code>	
<code>key_colnames</code>	Return the names of columns in the parent table that were used for grouping.
<code>keys</code>	

Methods Summary

<code>aggregate(func)</code>	Aggregate each group in the Table into a single row by applying the reduction function <code>func</code> to group values in each column.
<code>filter(func)</code>	Filter groups in the Table based on evaluating function <code>func</code> on each group sub-table.

Attributes Documentation

indices

key_colnames

Return the names of columns in the parent table that were used for grouping.

keys

Methods Documentation

aggregate (*func*)

Aggregate each group in the Table into a single row by applying the reduction function `func` to group values in each column.

Parameters

func : function

Function that reduces an array of values to a single value

Returns

out : Table

New table with the aggregated rows.

filter (*func*)

Filter groups in the Table based on evaluating function `func` on each group sub-table.

The function which is passed to this method must accept two arguments:

- `table`: Table object
- `key_colnames`: tuple of column names in `table` used as keys for grouping

It must then return either `True` or `False`. As an example, the following will select all table groups with only positive values in the non-key columns:

```
def all_positive(table, key_colnames):
    colnames = [name for name in table.colnames if name not in key_colnames]
    for colname in colnames:
        if np.any(table[colname] < 0):
            return False
    return True
```

Parameters

func: function
Filter function

Returns

out: Table
New table with the aggregated rows.

TableMergeError

exception `astropy.table.TableMergeError`

Class Inheritance Diagram

TIME AND DATES (ASTROPY.TIME)

9.1 Introduction

The `astropy.time` package provides functionality for manipulating times and dates. Specific emphasis is placed on supporting time scales (e.g. UTC, TAI, UT1, TDB) and time representations (e.g. JD, MJD, ISO 8601) that are used in astronomy and required to calculate, e.g., sidereal times and barycentric corrections. It uses Cython to wrap the C language `ERFA` time and calendar routines, using a fast and memory efficient vectorization scheme.

All time manipulations and arithmetic operations are done internally using two 64-bit floats to represent time. Floating point algorithms from¹ are used so that the `Time` object maintains sub-nanosecond precision over times spanning the age of the universe.

9.2 Getting Started

The basic way to use `astropy.time` is to create a `Time` object by supplying one or more input time values as well as the `time format` and `time scale` of those values. The input time(s) can either be a single scalar like "2010-01-01 00:00:00" or a list or a `numpy` array of values as shown below. In general any output values have the same shape (scalar or array) as the input.

```
>>> from astropy.time import Time
>>> times = ['1999-01-01T00:00:00.123456789', '2010-01-01T00:00:00']
>>> t = Time(times, format='isot', scale='utc')
>>> t
<Time object: scale='utc' format='isot' value=['1999-01-01T00:00:00.123' '2010-01-01T00:00:00.000']>
>>> t[1]
<Time object: scale='utc' format='isot' value=2010-01-01T00:00:00.000>
```

The `format` argument specifies how to interpret the input values, e.g. ISO or JD or Unix time. The `scale` argument specifies the `time scale` for the values, e.g. UTC or TT or UT1. The `scale` argument is optional and defaults to UTC except for `Time from epoch formats`. We could have written the above as:

```
>>> t = Time(times, format='isot')
```

When the format of the input can be unambiguously determined then the `format` argument is not required, so we can simplify even further:

```
>>> t = Time(times)
```

Now let's get the representation of these times in the JD and MJD formats by requesting the corresponding `Time` attributes:

¹ Shewchuk, 1997, *Discrete & Computational Geometry* 18(3):305-363

```
>>> t.jd
array([ 2451179.50000143, 2455197.5      ])
>>> t.mjd
array([ 51179.00000143, 55197.      ])
```

We can also convert to a different time scale, for instance from UTC to TT. This uses the same attribute mechanism as above but now returns a new `Time` object:

```
>>> t2 = t.tt
>>> t2
<Time object: scale='tt' format='isot' value=['1999-01-01T00:01:04.307' '2010-01-01T00:01:06.184']>
>>> t2.jd
array([ 2451179.5007443 , 2455197.50076602])
```

Note that both the ISO (ISOT) and JD representations of `t2` are different than for `t` because they are expressed relative to the TT time scale.

Finally, some further examples of what is possible. For details, see the API documentation below.

```
>>> dt = t[1] - t[0]
>>> dt
<TimeDelta object: scale='tai' format='jd' value=4018.0000217...>
```

Here, note the conversion of the timescale to TAI. Time differences can only have scales in which one day is always equal to 86400 seconds.

```
>>> import numpy as np
>>> t[0] + dt * np.linspace(0.,1.,12)
<Time object: scale='utc' format='isot' value=['1999-01-01T00:00:00.123' '2000-01-01T06:32:43.930'
'2000-12-31T13:05:27.737' '2001-12-31T19:38:11.544'
'2003-01-01T02:10:55.351' '2004-01-01T08:43:39.158'
'2004-12-31T15:16:22.965' '2005-12-31T21:49:06.772'
'2007-01-01T04:21:49.579' '2008-01-01T10:54:33.386'
'2008-12-31T17:27:17.193' '2010-01-01T00:00:00.000']>

>>> t.sidereal_time('apparent', 'greenwich')
<Longitude [ 6.68050179, 6.70281947] hourangle>
```

9.3 Using `astropy.time`

9.3.1 Time object basics

In `astropy.time` a “time” is a single instant of time which is independent of the way the time is represented (the “format”) and the time “scale” which specifies the offset and scaling relation of the unit of time. There is no distinction made between a “date” and a “time” since both concepts (as loosely defined in common usage) are just different representations of a moment in time.

Once a `Time` object is created it cannot be altered internally. In code lingo it is “immutable.” In particular the common operation of “converting” to a different `time scale` is always performed by returning a copy of the original `Time` object which has been converted to the new time scale.

Time Format

The time format specifies how an instant of time is represented. The currently available formats are can be found in the `Time.FORMATS` dict and are listed in the table below. Each of these formats is implemented as a class that derives

from the base `TimeFormat` class. This class structure can be easily adapted and extended by users for specialized time formats not supplied in `astropy.time`.

Format	Class	Example argument
<code>byear</code>	<code>TimeBesselianEpoch</code>	1950.0
<code>byear_str</code>	<code>TimeBesselianEpochString</code>	'B1950.0'
<code>cxsec</code>	<code>TimeCxcSec</code>	63072064.184
<code>datetime</code>	<code>TimeDatetime</code>	<code>datetime(2000, 1, 2, 12, 0, 0)</code>
<code>gps</code>	<code>TimeGPS</code>	630720013.0
<code>iso</code>	<code>TimeISO</code>	'2000-01-01 00:00:00.000'
<code>isot</code>	<code>TimeISOT</code>	'2000-01-01T00:00:00.000'
<code>jd</code>	<code>TimeJD</code>	2451544.5
<code>jyear</code>	<code>TimeJulianEpoch</code>	2000.0
<code>jyear_str</code>	<code>TimeJulianEpochString</code>	'J2000.0'
<code>mjd</code>	<code>TimeMJD</code>	51544.0
<code>plot_date</code>	<code>TimePlotDate</code>	730120.0003703703
<code>unix</code>	<code>TimeUnix</code>	946684800.0
<code>yday</code>	<code>TimeYearDayTime</code>	2000:001:00:00:00.000

Subformat

The time format classes `TimeISO`, `TimeISOT`, and `TimeYearDayTime` support the concept of subformats. This allows for variations on the basic theme of a format in both the input string parsing and the output.

The supported subformats are `date_hms`, `date_hm`, and `date`. The table below illustrates these subformats for `iso` and `yday` formats:

Format	Subformat	Input / output
<code>iso</code>	<code>date_hms</code>	2001-01-02 03:04:05.678
<code>iso</code>	<code>date_hm</code>	2001-01-02 03:04
<code>iso</code>	<code>date</code>	2001-01-02
<code>yday</code>	<code>date_hms</code>	2001:032:03:04:05.678
<code>yday</code>	<code>date_hm</code>	2001:032:03:04
<code>yday</code>	<code>date</code>	2001:032

Time from epoch formats

The formats `cxsec`, `gps`, and `unix` are a little special in that they provide a floating point representation of the elapsed time in seconds since a particular reference date. These formats have an intrinsic time scale which is used to compute the elapsed seconds since the reference date.

Format	Scale	Reference date
<code>cxsec</code>	TT	1998-01-01 00:00:00
<code>unix</code>	UTC	1970-01-01 00:00:00
<code>gps</code>	TAI	1980-01-06 00:00:19

Unlike the other formats which default to UTC, if no `scale` is provided when initializing a `Time` object then the above intrinsic scale is used. This is done for computational efficiency.

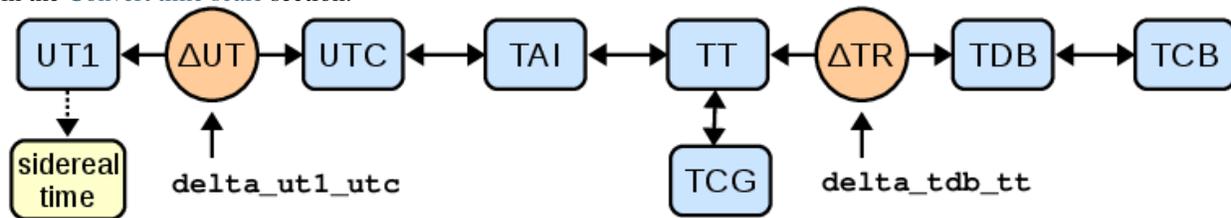
Time Scale

The time scale (or *time standard*) is “a specification for measuring time: either the rate at which time passes; or points in time; or both”². See also ³ and ⁴.

```
>>> Time.SCALES
('tai', 'tcb', 'tcg', 'tdb', 'tt', 'ut1', 'utc')
```

Scale	Description
tai	International Atomic Time (TAI)
tcb	Barycentric Coordinate Time (TCB)
tcg	Geocentric Coordinate Time (TCG)
tdb	Barycentric Dynamical Time (TDB)
tt	Terrestrial Time (TT)
ut1	Universal Time (UT1)
utc	Coordinated Universal Time (UTC)

The system of transformation between supported time scales is shown in the figure below. Further details are provided in the [Convert time scale](#) section.



Scalar or Array

A `Time` object can hold either a single time value or an array of time values. The distinction is made entirely by the form of the input time(s). If a `Time` object holds a single value then any format outputs will be a single scalar value, and likewise for arrays. Like other arrays and lists, `Time` objects holding arrays are subscriptable, returning scalar or array objects as appropriate:

```
>>> from astropy.time import Time
>>> t = Time(100.0, format='mjd')
>>> t.jd
2400100.5
>>> t = Time([100.0, 200.0, 300.], format='mjd')
>>> t.jd
array([ 2400100.5, 2400200.5, 2400300.5])
>>> t[:2]
<Time object: scale='utc' format='mjd' value=[ 100. 200.]>
>>> t[2]
<Time object: scale='utc' format='mjd' value=300.0>
```

Inferring input format

The `Time` class initializer will not accept ambiguous inputs, but it will make automatic inferences in cases where the inputs are unambiguous. This can apply when the times are supplied as `datetime` objects or strings. In the latter case it is not required to specify the format because the available string formats have no overlap. However, if the format is known in advance the string parsing will be faster if the format is provided.

² Wikipedia [time standard](#) article

³ SOFA [Time Scale and Calendar Tools \(PDF\)](#)

⁴ <http://www.ucolick.org/~sla/leapsecs/timescales.html>

```
>>> from datetime import datetime
>>> t = Time(datetime(2010, 1, 2, 1, 2, 3))
>>> t.format
'datetime'
>>> t = Time('2010-01-02 01:02:03')
>>> t.format
'iso'
```

Internal representation

The `Time` object maintains an internal representation of time as a pair of double precision numbers expressing Julian days. The sum of the two numbers is the Julian Date for that time relative to the given `time scale`. Users requiring no better than microsecond precision over human time scales (~100 years) can safely ignore the internal representation details and skip this section.

This representation is driven by the underlying ERFA C-library implementation. The ERFA routines take care throughout to maintain overall precision of the double pair. The user is free to choose the way in which total JD is provided, though internally one part contains integer days and the other the fraction of the day, as this ensures optimal accuracy for all conversions. The internal JD pair is available via the `jd1` and `jd2` attributes:

```
>>> t = Time('2010-01-01 00:00:00', scale='utc')
>>> t.jd1, t.jd2
(2455197.5, 0.0)
>>> t2 = t.tai
>>> t2.jd1, t2.jd2
(2455197.5, 0.00039351851851851...)
```

9.3.2 Creating a Time object

The allowed `Time` arguments to create a time object are listed below:

val

[numpy ndarray, list, str, or number] Data to initialize table.

val2

[numpy ndarray, list, str, or number; optional] Data to initialize table.

format

[str, optional] Format of input value(s)

scale

[str, optional] Time scale of input value(s)

precision

[int between 0 and 9 inclusive] Decimal precision when outputting seconds as floating point

in_subfmt

[str] Unix glob to select subformats for parsing string input times

out_subfmt

[str] Unix glob to select subformats for outputting string times

location

[`EarthLocation` or tuple, optional] If a tuple, 3 `Quantity` items with length units for geocentric coordinates, or a longitude, latitude, and optional height for geodetic coordinates. Can be a single location, or one for each input time.

val

The `val` argument specifies the input time or times and can be a single string or number, or it can be a Python list or `numpy` array of strings or numbers. To initialize a `Time` object based on a specified time, it *must* be present. If `val` is absent (or `None`), the `Time` object will be created for the time corresponding to the instant the object is created.

In most situations one also needs to specify the `time scale` via the `scale` argument. The `Time` class will never guess the `time scale`, so a simple example would be:

```
>>> t1 = Time(50100.0, scale='tt', format='mjd')
>>> t2 = Time('2010-01-01 00:00:00', scale='utc')
```

It is possible to create a new `Time` object from one or more existing time objects. In this case the format and scale will be inferred from the first object unless explicitly specified.

```
>>> Time([t1, t2])
<Time object: scale='tt' format='mjd' value=[ 50100. 55197.00076602]>
```

val2

The `val2` argument is available for specialized situations where extremely high precision is required. Recall that the internal representation of time within `astropy.time` is two double-precision numbers that when summed give the Julian date. If provided the `val2` argument is used in combination with `val` to set the second the internal time values. The exact interpretation of `val2` is determined by the input format class. As of this release all string-valued formats ignore `val2` and all numeric inputs effectively add the two values in a way that maintains the highest precision. Example:

```
>>> t = Time(100.0, 0.000001, format='mjd', scale='tt')
>>> t.jd, t.jd1, t.jd2
(2400100.50000..., 2400100.5, 1e-06)
```

format

The `format` argument sets the `time format`, and as mentioned it is required unless the format can be unambiguously determined from the input times.

scale

The `scale` argument sets the `time scale` and is required except for time formats such as `plot_date` (`TimePlotDate`) and `unix` (`TimeUnix`). These formats represent the duration in SI seconds since a fixed instant in time which is independent of time scale.

precision

The `precision` setting affects string formats when outputting a value that includes seconds. It must be an integer between 0 and 9. There is no effect when inputting time values from strings. The default precision is 3. Note that the limit of 9 digits is driven by the way that ERFA handles fractional seconds. In practice this should not be an issue.

```
>>> t = Time('B1950.0', scale='utc', precision=3)
>>> t.byyear_str
'B1950.000'
>>> t.precision = 0
```

```
>>> t.byyear_str
'B1950'
```

in_subfmt

The `in_subfmt` argument provides a mechanism to select one or more `subformat` values from the available subformats for string input. Multiple allowed subformats can be selected using Unix-style wildcard characters, in particular `*` and `?`, as documented in the Python `fnmatch` module.

The default value for `in_subfmt` is `*` which matches any available subformat. This allows for convenient input of values with unknown or heterogeneous subformat:

```
>>> Time(['2000:001', '2000:002:03:04', '2001:003:04:05:06.789'])
<Time object: scale='utc' format='yday'
  value=['2000:001:00:00:00.000' '2000:002:03:04:00.000' '2001:003:04:05:06.789']>
```

One can explicitly specify `in_subfmt` in order to strictly require a certain subformat:

```
>>> t = Time('2000:002:03:04', in_subfmt='date_hm')
>>> t = Time('2000:002', in_subfmt='date_hm')
Traceback (most recent call last):
...
ValueError: Input values did not match any of the formats where the
format keyword is optional ['astropy_time', 'datetime',
'byear_str', 'iso', 'isot', 'jyear_str', 'yday']
Traceback (most recent call last):
...
ValueError: Input values did not match any of the formats where the
```

out_subfmt

The `out_subfmt` argument is similar to `in_subfmt` except that it applies to output formatting. In the case of multiple matching subformats the first matching subformat is used.

```
>>> Time('2000-01-01 02:03:04', out_subfmt='date').iso
'2000-01-01'
>>> Time('2000-01-01 02:03:04', out_subfmt='date_hms').iso
'2000-01-01 02:03:04.000'
>>> Time('2000-01-01 02:03:04', out_subfmt='date*').iso
'2000-01-01 02:03:04.000'
```

location

This optional parameter specifies the observer location, using an `EarthLocation` object or a tuple containing any form that can initialize one: either a tuple with geocentric coordinates (X, Y, Z), or a tuple with geodetic coordinates (longitude, latitude, height; with height defaulting to zero). They are used for time scales that are sensitive to observer location (currently, only TDB, which relies on the ERFA routine `eraDtdb` to determine the time offset between TDB and TT), as well as for sidereal time if no explicit longitude is given.

```
>>> t = Time('2001-03-22 00:01:44.732327132980', scale='utc',
...         location=('120d', '40d'))
>>> t.sidereal_time('apparent', 'greenwich')
<Longitude 12.000000000000... hourangle>
>>> t.sidereal_time('apparent')
<Longitude 20.000000000000... hourangle>
```

Note: In future versions, we hope to add the possibility to add observatory objects and/or names.

Getting the Current Time

The current time can be determined as a `Time` object using the `now` class method:

```
>>> nt = Time.now()
>>> ut = Time(datetime.utcnow(), scale='utc')
```

The two should be very close to each other.

9.3.3 Using Time objects

There are four basic operations available with `Time` objects:

- Get the representation of the time value(s) in a particular `time format`.
- Get a new time object for the same time value(s) but referenced to a different `time scale`.
- Calculate the `sidereal time` corresponding to the time value(s).
- Do time arithmetic involving `Time` and/or `TimeDelta` objects.

Get representation

Instants of time can be represented in different ways, for instance as an ISO-format date string ('1999-07-23 04:31:00') or seconds since 1998.0 (49091460.0) or Modified Julian Date (51382.187451574).

The representation of a `Time` object in a particular format is available by getting the object attribute corresponding to the format name. The list of available format names is in the `time format` section.

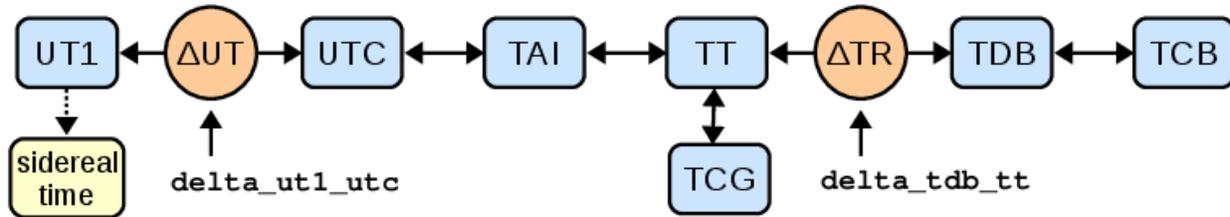
```
>>> t = Time('2010-01-01 00:00:00', format='iso', scale='utc')
>>> t.jd          # JD representation of time in current scale (UTC)
2455197.5
>>> t.iso        # ISO representation of time in current scale (UTC)
'2010-01-01 00:00:00.000'
>>> t.unix       # seconds since 1970.0 (UTC)
1262304000.0
>>> t.plot_date # Date value for plotting with matplotlib plot_date()
733773.0
>>> t.datetime  # Representation as datetime.datetime object
datetime.datetime(2010, 1, 1, 0, 0)
```

Example:

```
>>> import matplotlib.pyplot as plt
>>> jyear = np.linspace(2000, 2001, 20)
>>> t = Time(jyear, format='jyear')
>>> plt.plot_date(t.plot_date, jyear)
>>> plt.gcf().autofmt_xdate() # orient date labels at a slant
>>> plt.draw()
```

Convert time scale

A new `Time` object for the same time value(s) but referenced to a new `time scale` can be created getting the object attribute corresponding to the time scale name. The list of available time scale names is in the `time scale` section and in the figure below illustrating the network of time scale transformations.



Examples:

```
>>> t = Time('2010-01-01 00:00:00', format='iso', scale='utc')
>>> t.tt           # TT scale
<Time object: scale='tt' format='iso' value=2010-01-01 00:01:06.184>
>>> t.tai
<Time object: scale='tai' format='iso' value=2010-01-01 00:00:34.000>
```

In this process the `format` and other object attributes like `lon`, `lat`, and `precision` are also propagated to the new object.

As noted in the `Time object basics` section, a `Time` object is immutable and the internal time values cannot be altered once the object is created. The process of changing the time scale therefore begins by making a copy of the original object and then converting the internal time values in the copy to the new time scale. The new `Time` object is returned by the attribute access.

Transformation offsets

Time scale transformations that cross one of the orange circles in the image above require an additional offset time value that is model or observation-dependent. See [SOFA Time Scale and Calendar Tools](#) for further details.

The two attributes `delta_ut1_utc` and `delta_tdb_tt` provide a way to set these offset times explicitly. These represent the time scale offsets UT1 - UTC and TDB - TT, respectively. As an example:

```
>>> t = Time('2010-01-01 00:00:00', format='iso', scale='utc')
>>> t.delta_ut1_utc = 0.334 # Explicitly set one part of the transformation
>>> t.ut1.iso         # ISO representation of time in UT1 scale
'2010-01-01 00:00:00.334'
```

For the UT1 to UTC offset, one has to interpolate in observed values provided by the [International Earth Rotation and Reference Systems Service](#). By default, Astropy is shipped with the final values provided in [Bulletin B](#), which cover the period from 1962 to shortly before an astropy release, and these will be used to compute the offset if the `delta_ut1_utc` attribute is not set explicitly. For more recent times, one can download an updated version of [IERS B](#) or [IERS A](#) (which also has predictions), and set `delta_ut1_utc` as described in `get_delta_ut1_utc`:

```
>>> from astropy.utils.iers import IERS_A, IERS_A_URL
>>> from astropy.utils.data import download_file
>>> iers_a_file = download_file(IERS_A_URL, cache=True)
>>> iers_a = IERS_A.open(iers_a_file)
>>> t.delta_ut1_utc = t.get_delta_ut1_utc(iers_a)
```

In the case of the TDB to TT offset, most users need only provide the `lon` and `lat` values when creating the `Time` object. If the `delta_tdb_tt` attribute is not explicitly set then the ERFA C-library routine `eraDtdb` will be used

to compute the TDB to TT offset. Note that if `lon` and `lat` are not explicitly initialized, values of 0.0 degrees for both will be used.

The following code replicates an example in the [SOFA Time Scale and Calendar Tools](#) document. It does the transform from UTC to all supported time scales (TAI, TCB, TCG, TDB, TT, UT1, UTC). This requires an observer location (here, latitude and longitude):

```
>>> import astropy.units as u
>>> t = Time('2006-01-15 21:24:37.5', format='iso', scale='utc',
...         location=(-155.933222*u.deg, 19.48125*u.deg), precision=6)
>>> t.utc.iso
'2006-01-15 21:24:37.500000'
>>> t.ut1.iso
'2006-01-15 21:24:37.834078'
>>> t.tai.iso
'2006-01-15 21:25:10.500000'
>>> t.tt.iso
'2006-01-15 21:25:42.684000'
>>> t.tcg.iso
'2006-01-15 21:25:43.322690'
>>> t.tdb.iso
'2006-01-15 21:25:42.684373'
>>> t.tcb.iso
'2006-01-15 21:25:56.893952'
```

9.3.4 Sidereal Time

Apparent or mean sidereal time can be calculated using `sidereal_time()`. The method returns a `Longitude` with units of hourangle, which by default is for the longitude corresponding to the location with which the `Time` object is initialized. Like the scale transformations, ERFA C-library routines are used under the hood, which support calculations following different IAU resolutions. Sample usage:

```
>>> t = Time('2006-01-15 21:24:37.5', scale='utc', location=('120d', '45d'))
>>> t.sidereal_time('mean')
<Longitude 13.089521870640... hourangle>
>>> t.sidereal_time('apparent')
<Longitude 13.08950367508... hourangle>
>>> t.sidereal_time('apparent', 'greenwich')
<Longitude 5.08950367508... hourangle>
>>> t.sidereal_time('apparent', '-90d')
<Longitude 23.08950367508... hourangle>
>>> t.sidereal_time('apparent', '-90d', 'IAU1994')
<Longitude 23.08950365423... hourangle>
```

9.3.5 Time Deltas

Simple time arithmetic is supported using the `TimeDelta` class. The following operations are available:

- Create a `TimeDelta` explicitly by instantiating a class object
- Create a `TimeDelta` by subtracting two `Times`
- Add a `TimeDelta` to a `Time` object to get a new `Time`
- Subtract a `TimeDelta` from a `Time` object to get a new `Time`
- Add two `TimeDelta` objects to get a new `TimeDelta`

- Negate a `TimeDelta` or take its absolute value
- Multiply or divide a `TimeDelta` by a constant or array
- Convert `TimeDelta` objects to and from time-like Quantities

The `TimeDelta` class is derived from the `Time` class and shares many of its properties. One difference is that the time scale has to be one for which one day is exactly 86400 seconds. Hence, the scale cannot be UTC.

The available time formats are:

Format	Class
sec	<code>TimeDeltaSec</code>
jd	<code>TimeDeltaJD</code>

Examples

Use of the `TimeDelta` object is easily illustrated in the few examples below:

```
>>> t1 = Time('2010-01-01 00:00:00')
>>> t2 = Time('2010-02-01 00:00:00')
>>> dt = t2 - t1 # Difference between two Times
>>> dt
<TimeDelta object: scale='tai' format='jd' value=31.0>
>>> dt.sec
2678400.0

>>> from astropy.time import TimeDelta
>>> dt2 = TimeDelta(50.0, format='sec')
>>> t3 = t2 + dt2 # Add a TimeDelta to a Time
>>> t3.iso
'2010-02-01 00:00:50.000'

>>> t2 - dt2 # Subtract a TimeDelta from a Time
<Time object: scale='utc' format='iso' value=2010-01-31 23:59:10.000>

>>> dt + dt2
<TimeDelta object: scale='tai' format='jd' value=31.0005787037>

>>> import numpy as np
>>> t1 + dt * np.linspace(0, 1, 5)
<Time object: scale='utc' format='iso' value=['2010-01-01 00:00:00.000'
'2010-01-08 18:00:00.000' '2010-01-16 12:00:00.000' '2010-01-24 06:00:00.000'
'2010-02-01 00:00:00.000']>
```

9.3.6 Time Scales for Time Deltas

Above, one sees that the difference between two UTC times is a `TimeDelta` with a scale of TAI. This is because a UTC time difference cannot be uniquely defined unless one knows the two times that were differenced (because of leap seconds, a day does not always have 86400 seconds). For all other time scales, the `TimeDelta` inherits the scale of the first `Time` object:

```
>>> t1 = Time('2010-01-01 00:00:00', scale='tcg')
>>> t2 = Time('2011-01-01 00:00:00', scale='tcg')
>>> dt = t2 - t1
>>> dt
<TimeDelta object: scale='tcg' format='jd' value=365.0>
```

When `TimeDelta` objects are added or subtracted from `Time` objects, scales are converted appropriately, with the final scale being that of the `Time` object:

```
>>> t2 + dt
<Time object: scale='tcg' format='iso' value=2012-01-01 00:00:00.000>
>>> t2.tai
<Time object: scale='tai' format='iso' value=2010-12-31 23:59:27.068>
>>> t2.tai + dt
<Time object: scale='tai' format='iso' value=2011-12-31 23:59:27.046>
```

`TimeDelta` objects can be converted only to objects with compatible scales, i.e., scales for which it is not necessary to know the times that were differenced:

```
>>> dt.tt
<TimeDelta object: scale='tt' format='jd' value=364.99999974...>
>>> dt.tdb
Traceback (most recent call last):
...
ScaleValueError: Cannot convert TimeDelta with scale 'tcg' to scale 'tdb'
Traceback (most recent call last):
...
ScaleValueError: Cannot convert TimeDelta with scale 'tcg' to scale 'tdb'
```

`TimeDelta` objects can also have an undefined scale, in which case it is assumed that their scale matches that of the other `Time` or `TimeDelta` object (or is TAI in case of a UTC time):

```
>>> t2.tai + TimeDelta(365., format='jd', scale=None)
<Time object: scale='tai' format='iso' value=2011-12-31 23:59:27.068>
```

9.3.7 Interaction with Time-like Quantities

Where possible, `Quantity` objects with units of time are treated as `TimeDelta` objects with undefined scale (though necessarily with lower precision). They can also be used as input in constructing `Time` and `TimeDelta` objects, and `TimeDelta` objects can be converted to `Quantity` objects of arbitrary units of time. Usage is most easily illustrated by examples:

```
>>> import astropy.units as u
>>> Time(10.*u.yr, format='gps') # time-valued quantities can be used for
...                             # for formats requiring a time offset
<Time object: scale='tai' format='gps' value=315576000.0>
>>> Time(10.*u.yr, 1.*u.s, format='gps')
<Time object: scale='tai' format='gps' value=315576001.0>
>>> Time(2000.*u.yr, scale='utc', format='jyear')
<Time object: scale='utc' format='jyear' value=2000.0>
>>> Time(2000.*u.yr, scale='utc', format='byear')
...                             # but not for Besselian year, which implies
...                             # a different time scale
...
Traceback (most recent call last):
...
ValueError: Input values did not match the format class byear

>>> TimeDelta(10.*u.yr) # With a quantity, no format is required
<TimeDelta object: scale='None' format='jd' value=3652.5>

>>> dt = TimeDelta([10., 20., 30.], format='jd')
>>> dt.to(u.hr) # can convert TimeDelta to a quantity
<Quantity [ 240., 480., 720.] h>
```

```

>>> dt > 400. * u.hr # and compare to quantities with units of time
array([False,  True,  True], dtype=bool)
>>> dt + 1.*u.hr # can also add/subtract such quantities
<TimeDelta object: scale='None' format='jd' value=[ 10.04166667  20.04166667  30.04166667]>
>>> Time(50000., format='mjd', scale='utc') + 1.*u.hr
<Time object: scale='utc' format='mjd' value=50000.041666...>
>>> dt * 10.*u.km/u.s # for multiplication and division with a
... # Quantity, TimeDelta is converted
<Quantity [ 100., 200., 300.] d km / s>
>>> dt * 10.*u.Unit(1) # unless the Quantity is dimensionless
<TimeDelta object: scale='None' format='jd' value=[ 100.  200.  300.]>
Traceback (most recent call last):
...
ValueError: Input values did not match the format class byear

```

9.4 Reference/API

9.4.1 astropy.time Module

Classes

<code>OperandTypeError(left, right[, op])</code>	
<code>ScaleValueError</code>	
<code>Time(*args, **kwargs)</code>	Represent and manipulate times and dates for astronomy.
<code>TimeBesselianEpoch(val1, val2, scale, ...[, ...])</code>	Besselian Epoch year as floating point value(s) like 1950.0
<code>TimeBesselianEpochString(val1, val2, scale, ...)</code>	Besselian Epoch year as string value(s) like 'B1950.0'
<code>TimeCxcSec(val1, val2, scale, precision, ...)</code>	Chandra X-ray Center seconds from 1998-01-01 00:00:00 TT.
<code>TimeDatetime(val1, val2, scale, precision, ...)</code>	Represent date as Python standard library <code>datetime</code> object
<code>TimeDelta(val[, val2, format, scale, copy])</code>	Represent the time difference between two times.
<code>TimeDeltaFormat(val1, val2, scale, ...[, ...])</code>	Base class for time delta representations
<code>TimeDeltaJD(val1, val2, scale, precision, ...)</code>	Time delta in Julian days (86400 SI seconds)
<code>TimeDeltaSec(val1, val2, scale, precision, ...)</code>	Time delta in SI seconds
<code>TimeEpochDate(val1, val2, scale, precision, ...)</code>	Base class for support floating point Besselian and Julian epoch dates
<code>TimeEpochDateString(val1, val2, scale, ...)</code>	Base class to support string Besselian and Julian epoch dates such as 'B1950.0'
<code>TimeFormat(val1, val2, scale, precision, ...)</code>	Base class for time representations.
<code>TimeFromEpoch(val1, val2, scale, precision, ...)</code>	Base class for times that represent the interval from a particular epoch as a
<code>TimeGPS(val1, val2, scale, precision, ...[, ...])</code>	GPS time: seconds from 1980-01-06 00:00:00 UTC For example, 630720000
<code>TimeISO(val1, val2, scale, precision, ...[, ...])</code>	ISO 8601 compliant date-time format "YYYY-MM-DD HH:MM:SS.sss..."
<code>TimeISOT(val1, val2, scale, precision, ...)</code>	ISO 8601 compliant date-time format "YYYY-MM-DDTHH:MM:SS.sss..."
<code>TimeJD(val1, val2, scale, precision, ...[, ...])</code>	Julian Date time format.
<code>TimeJulianEpoch(val1, val2, scale, ...[, ...])</code>	Julian Epoch year as floating point value(s) like 2000.0
<code>TimeJulianEpochString(val1, val2, scale, ...)</code>	Julian Epoch year as string value(s) like 'J2000.0'
<code>TimeMJD(val1, val2, scale, precision, ...[, ...])</code>	Modified Julian Date time format.
<code>TimePlotDate(val1, val2, scale, precision, ...)</code>	Matplotlib <code>plot_date</code> input: 1 + number of days from 0001-01-01 00:00:00
<code>TimeString(val1, val2, scale, precision, ...)</code>	Base class for string-like time representations.
<code>TimeUnix(val1, val2, scale, precision, ...)</code>	Unix time: seconds from 1970-01-01 00:00:00 UTC.
<code>TimeYearDayTime(val1, val2, scale, ...[, ...])</code>	Year, day-of-year and time as "YYYY:DOY:HH:MM:SS.sss..."

OperandTypeError

exception `astropy.time.OperandTypeError` (*left, right, op=None*)

ScaleValueError

exception `astropy.time.ScaleValueError`

Time

class `astropy.time.Time` (**args, **kwargs*)

Bases: `object`

Represent and manipulate times and dates for astronomy.

A `Time` object is initialized with one or more times in the `val` argument. The input times in `val` must conform to the specified `format` and must correspond to the specified time `scale`. The optional `val2` time input should be supplied only for numeric input formats (e.g. JD) where very high precision (better than 64-bit precision) is required.

Parameters

val : sequence, str, number, or `Time` object

Value(s) to initialize the time or times.

val2 : sequence, str, or number; optional

Value(s) to initialize the time or times.

format : str, optional

Format of input value(s)

scale : str, optional

Time scale of input value(s)

precision : int, optional

Digits of precision in string representation of time

in_subfmt : str, optional

Subformat for inputting string times

out_subfmt : str, optional

Subformat for outputting string times

location : `EarthLocation` or tuple, optional

If given as an tuple, it should be able to initialize an `EarthLocation` instance, i.e., either contain 3 items with units of length for geocentric coordinates, or contain a longitude, latitude, and an optional height for geodetic coordinates. Can be a single location, or one for each input time.

copy : bool, optional

Make a copy of the input values

Attributes Summary

FORMATS	
SCALES	tuple() -> empty tuple
delta_tdb_tt	
delta_ut1_utc	Get ERFA DUT arg = UT1 - UTC.
format	Time format
in_subfmt	Unix wildcard pattern to select subformats for parsing string input times.
is_scalar	Deprecated since version 0.3.
jd1	First of the two doubles that internally store time value(s) in JD.
jd2	Second of the two doubles that internally store time value(s) in JD.
lat	Deprecated since version 0.4.
lon	Deprecated since version 0.4.
out_subfmt	Unix wildcard pattern to select subformats for outputting times.
precision	Decimal precision when outputting seconds as floating point (int value between 0 and 9 inclusive).
scale	Time scale
val	Deprecated since version 0.3.
vals	Deprecated since version 0.3.
value	Time value(s) in current format

Methods Summary

<code>copy([format])</code>	Return a fully independent copy the Time object, optionally changing the format.
<code>get_delta_ut1_utc([iers_table, return_status])</code>	Find UT1 - UTC differences by interpolating in IERS Table.
<code>now()</code>	Creates a new object corresponding to the instant in time this method is called.
<code>replicate([format, copy])</code>	Return a replica of the Time object, optionally changing the format.
<code>sidereal_time(kind[, longitude, model])</code>	Calculate sidereal time.

Attributes Documentation

FORMATS = {u'mjd': <class 'astropy.time.core.TimeMJD'>, u'excsec': <class 'astropy.time.core.TimeCxcSec'>, u'jyear':
Dict of time formats

SCALES = (u'tai', u'tcb', u'tcg', u'tdb', u'tt', u'ut1', u'utc')
List of time scales

delta_tdb_tt
TDB - TT time scale offset

delta_ut1_utc
UT1 - UTC time scale offset

format
Time format

in_subfmt
Unix wildcard pattern to select subformats for parsing string input times.

is_scalar
Deprecated since version 0.3: The is_scalar attribute is deprecated and may be removed in a future version.

jd1
First of the two doubles that internally store time value(s) in JD.

jd2

Second of the two doubles that internally store time value(s) in JD.

lat

Deprecated since version 0.4: The lat function is deprecated and may be removed in a future version. Use `location.latitude` instead.

lon

Deprecated since version 0.4: The lon function is deprecated and may be removed in a future version. Use `location.longitude` instead.

out_subfmt

Unix wildcard pattern to select subformats for outputting times.

precision

Decimal precision when outputting seconds as floating point (int value between 0 and 9 inclusive).

scale

Time scale

val

Deprecated since version 0.3: The val function is deprecated and may be removed in a future version. Use `value` instead.

vals

Deprecated since version 0.3: The vals function is deprecated and may be removed in a future version. Use `value` instead.

Time values in current format as a numpy array.

value

Time value(s) in current format

Methods Documentation

copy (*format=None*)

Return a fully independent copy the Time object, optionally changing the format.

If `format` is supplied then the time format of the returned Time object will be set accordingly, otherwise it will be unchanged from the original.

In this method a full copy of the internal time arrays will be made. The internal time arrays are normally not changeable by the user so in most cases the `replicate()` method should be used.

Parameters

format : str, optional

Time format of the copy.

Returns

tm: Time object

Copy of this object

get_delta_ut1_utc (*iers_table=None, return_status=False*)

Find UT1 - UTC differences by interpolating in IERS Table.

Parameters

iers_table : `astropy.utils.iers.IERS` table, optional

Table containing UT1-UTC differences from IERS Bulletins A and/or B. If `None`, use default version (see `astropy.utils.iers`)

return_status : bool

Whether to return status values. If `False` (default), `iers` raises `IndexError` if any time is out of the range covered by the IERS table.

Returns

`ut1_utc`: float or float array

UT1-UTC, interpolated in IERS Table

`status`: int or int array

Status values (if `return_status='True'`): `astropy.utils.iers.FROM_IERS_B`
`astropy.utils.iers.FROM_IERS_A` `astropy.utils.iers.FROM_IERS_A_PREDICTION`
`astropy.utils.iers.TIME_BEFORE_IERS_RANGE`
`astropy.utils.iers.TIME_BEYOND_IERS_RANGE`

Notes

In normal usage, UT1-UTC differences are calculated automatically on the first instance `ut1` is needed.

Examples

To check in code whether any times are before the IERS table range:

```
>>> from astropy.utils.iers import TIME_BEFORE_IERS_RANGE
>>> t = Time(['1961-01-01', '2000-01-01'], scale='utc')
>>> delta, status = t.get_delta_ut1_utc(return_status=True)
>>> status == TIME_BEFORE_IERS_RANGE
array([ True, False], dtype=bool)
```

To use an updated IERS A bulletin to calculate UT1-UTC (see also `astropy.utils.iers`):

```
>>> from astropy.utils.iers import IERS_A, IERS_A_URL
>>> from astropy.utils.data import download_file
>>> t = Time(['1974-01-01', '2000-01-01'], scale='utc')
>>> iers_a_file = download_file(IERS_A_URL,
...                             cache=True)
Downloading ... [Done]
>>> iers_a = IERS_A.open(iers_a_file)
>>> t.delta_ut1_utc = t.get_delta_ut1_utc(iers_a)
```

The `delta_ut1_utc` property will be used to calculate `t.ut1`; raises `IndexError` if any of the times is out of range.

classmethod `now()`

Creates a new object corresponding to the instant in time this method is called.

Note: “Now” is determined using the `utcnow` function, so its accuracy and precision is determined by that function. Generally that means it is set by the accuracy of your system clock.

Returns

`nowtime`

A new `Time` object (or a subclass of `Time` if this is called from such a subclass) at the current time.

replicate (*format=None, copy=False*)

Return a replica of the Time object, optionally changing the format.

If `format` is supplied then the time format of the returned Time object will be set accordingly, otherwise it will be unchanged from the original.

If `copy` is set to `True` then a full copy of the internal time arrays will be made. By default the replica will use a reference to the original arrays when possible to save memory. The internal time arrays are normally not changeable by the user so in most cases it should not be necessary to set `copy` to `True`.

The convenience method `copy()` is available in which `copy` is `True` by default.

Parameters

format : str, optional

Time format of the replica.

copy : bool, optional

Return a true copy instead of using references where possible.

Returns

tm: Time object

Replica of this object

sidereal_time (*kind, longitude=None, model=None*)

Calculate sidereal time.

Parameters

kind : str

'mean' or 'apparent', i.e., accounting for precession only, or also for nutation.

longitude : `Quantity`, str, or `None`; optional

The longitude on the Earth at which to compute the sidereal time.

Can be given as a `Quantity` with angular units (or an `Angle` or `Longitude`), or as a name of an observatory (currently, only 'greenwich' is supported, equivalent to 0 deg). If `None` (default), the `lon` attribute of the Time object is used.

model : str or `None`; optional

Precession (and nutation) model to use. The available ones are: - apparent: [u'IAU1994', u'IAU2000A', u'IAU2000B', u'IAU2006A'] - mean: [u'IAU1982', u'IAU2000', u'IAU2006'] If `None` (default), the last (most recent) one from the appropriate list above is used.

Returns

sidereal time : `Longitude`

Sidereal time as a quantity with units of hourangle

TimeBesselianEpoch

class `astropy.time.TimeBesselianEpoch` (*val1, val2, scale, precision, in_subfmt, out_subfmt, from_jd=False*)

Bases: `astropy.time.TimeEpochDate`

Besselian Epoch year as floating point value(s) like 1950.0

Attributes Summary

<code>epoch_to_jd</code>	<code>unicode(string [, encoding[, errors]]) -> object</code>
<code>jd_to_epoch</code>	<code>unicode(string [, encoding[, errors]]) -> object</code>
<code>name</code>	<code>unicode(string [, encoding[, errors]]) -> object</code>

Attributes Documentation

`epoch_to_jd = u'besselian_epoch_jd'`

`jd_to_epoch = u'jd_besselian_epoch'`

`name = u'byear'`

TimeBesselianEpochString

`class astropy.time.TimeBesselianEpochString` (*val1, val2, scale, precision, in_subfmt, out_subfmt, from_jd=False*)

Bases: `astropy.time.TimeEpochDateString`

Besselian Epoch year as string value(s) like 'B1950.0'

Attributes Summary

<code>epoch_prefix</code>	<code>unicode(string [, encoding[, errors]]) -> object</code>
<code>epoch_to_jd</code>	<code>unicode(string [, encoding[, errors]]) -> object</code>
<code>jd_to_epoch</code>	<code>unicode(string [, encoding[, errors]]) -> object</code>
<code>name</code>	<code>unicode(string [, encoding[, errors]]) -> object</code>

Attributes Documentation

`epoch_prefix = u'B'`

`epoch_to_jd = u'besselian_epoch_jd'`

`jd_to_epoch = u'jd_besselian_epoch'`

`name = u'byear_str'`

TimeCxcSec

`class astropy.time.TimeCxcSec` (*val1, val2, scale, precision, in_subfmt, out_subfmt, from_jd=False*)

Bases: `astropy.time.TimeFromEpoch`

Chandra X-ray Center seconds from 1998-01-01 00:00:00 TT. For example, 63072064.184 is midnight on January 1, 2000.

Attributes Summary

<code>epoch_format</code>	<code>unicode(string [, encoding[, errors]]) -> object</code>
<code>epoch_scale</code>	<code>unicode(string [, encoding[, errors]]) -> object</code>
<code>epoch_val</code>	<code>unicode(string [, encoding[, errors]]) -> object</code>
<code>epoch_val2</code>	
<code>name</code>	<code>unicode(string [, encoding[, errors]]) -> object</code>
<code>unit</code>	<code>float(x) -> floating point number</code>

Attributes Documentation

`epoch_format = u'iso'`

`epoch_scale = u'tt'`

`epoch_val = u'1998-01-01 00:00:00'`

`epoch_val2 = None`

`name = u'cxsec'`

`unit = 1.1574074074074073e-05`

TimeDatetime

`class astropy.time.TimeDatetime` (*val1, val2, scale, precision, in_subfmt, out_subfmt, from_jd=False*)

Bases: `astropy.time.core.TimeUnique`

Represent date as Python standard library `datetime` object

Example:

```
>>> from datetime import datetime
>>> t = Time(datetime(2000, 1, 2, 12, 0, 0), scale='utc')
>>> t.iso
'2000-01-02 12:00:00.000'
>>> t.tt.datetime
datetime.datetime(2000, 1, 2, 12, 1, 4, 184000)
```

Attributes Summary

<code>name</code>	<code>unicode(string [, encoding[, errors]]) -> object</code>
<code>value</code>	

Methods Summary

`set_jds(val1, val2)` Convert datetime object contained in val1 to jd1, jd2

Attributes Documentation

name = u'datetime'

value

Methods Documentation

set_jds (*val1, val2*)
Convert datetime object contained in val1 to jd1, jd2

TimeDelta

class `astropy.time.TimeDelta` (*val, val2=None, format=None, scale=None, copy=False*)
Bases: `astropy.time.Time`

Represent the time difference between two times.

A `TimeDelta` object is initialized with one or more times in the `val` argument. The input times in `val` must conform to the specified `format`. The optional `val2` time input should be supplied only for numeric input formats (e.g. JD) where very high precision (better than 64-bit precision) is required.

Parameters

val : numpy ndarray, list, str, number, or `TimeDelta` object

Data to initialize table.

val2 : numpy ndarray, list, str, or number; optional

Data to initialize table.

format : str, optional

Format of input value(s)

scale : str, optional

Time scale of input value(s)

copy : bool, optional

Make a copy of the input values

Attributes Summary

FORMATS
SCALES `list()` -> new empty list

Methods Summary

```
replicate(*args, **kwargs)
to(*args, **kwargs)
```

Attributes Documentation

FORMATS = {**u'jd'**: <class 'astropy.time.core.TimeDeltaJD'>, **u'sec'**: <class 'astropy.time.core.TimeDeltaSec'>}
Dict of time delta formats.

SCALES = [**u'tdb'**, **u'tt'**, **u'ut1'**, **u'tcg'**, **u'tcb'**, **u'tai'**]
List of time delta scales.

Methods Documentation

replicate (*args, **kwargs)

to (*args, **kwargs)

TimeDeltaFormat

class astropy.time.**TimeDeltaFormat** (val1, val2, scale, precision, in_subfmt, out_subfmt, from_jd=False)

Bases: astropy.time.TimeFormat

Base class for time delta representations

Attributes Summary

```
value
```

Methods Summary

```
set_jds(val1, val2)
```

Attributes Documentation

value

Methods Documentation

set_jds (val1, val2)

TimeDeltaJD

class `astropy.time.TimeDeltaJD` (*val1, val2, scale, precision, in_subfmt, out_subfmt, from_jd=False*)

Bases: `astropy.time.TimeDeltaFormat`

Time delta in Julian days (86400 SI seconds)

Attributes Summary

<code>name</code>	<code>unicode(string [, encoding[, errors]]) -> object</code>
<code>unit</code>	<code>float(x) -> floating point number</code>

Attributes Documentation

`name = u'jd'`

`unit = 1.0`

TimeDeltaSec

class `astropy.time.TimeDeltaSec` (*val1, val2, scale, precision, in_subfmt, out_subfmt, from_jd=False*)

Bases: `astropy.time.TimeDeltaFormat`

Time delta in SI seconds

Attributes Summary

<code>name</code>	<code>unicode(string [, encoding[, errors]]) -> object</code>
<code>unit</code>	<code>float(x) -> floating point number</code>

Attributes Documentation

`name = u'sec'`

`unit = 1.1574074074074073e-05`

TimeEpochDate

class `astropy.time.TimeEpochDate` (*val1, val2, scale, precision, in_subfmt, out_subfmt, from_jd=False*)

Bases: `astropy.time.TimeFormat`

Base class for support floating point Besselian and Julian epoch dates

Attributes Summary

`value`

Methods Summary

`set_jds(val1, val2)`

Attributes Documentation

value

Methods Documentation

set_jds (*val1*, *val2*)

TimeEpochDateString

class `astropy.time.TimeEpochDateString` (*val1*, *val2*, *scale*, *precision*, *in_subfmt*, *out_subfmt*,
from_jd=False)

Bases: `astropy.time.TimeString`

Base class to support string Besselian and Julian epoch dates such as 'B1950.0' or 'J2000.0' respectively.

Attributes Summary

`value`

Methods Summary

`set_jds(val1, val2)`

Attributes Documentation

value

Methods Documentation

set_jds (*val1*, *val2*)

TimeFormat

class `astropy.time.TimeFormat` (*val1, val2, scale, precision, in_subfmt, out_subfmt, from_jd=False*)

Bases: `object`

Base class for time representations.

Parameters

val1 : numpy ndarray, list, str, or number

Data to initialize table.

val2 : numpy ndarray, list, str, or number; optional

Data to initialize table.

scale : str

Time scale of input value(s)

precision : int

Precision for seconds as floating point

in_subfmt : str

Select subformat for inputting string times

out_subfmt : str

Select subformat for outputting string times

from_jd : bool

If true then val1, val2 are jd1, jd2

Attributes Summary

<code>scale</code>	Time scale
<code>value</code>	Return time representation from internal jd1 and jd2.

Methods Summary

<code>set_jds(val1, val2)</code>	Set internal jd1 and jd2 from val1 and val2.
----------------------------------	--

Attributes Documentation

scale

Time scale

value

Return time representation from internal jd1 and jd2. Must be provided by derived classes.

Methods Documentation

set_jds (*val1, val2*)

Set internal jd1 and jd2 from val1 and val2. Must be provided by derived classes.

TimeFromEpoch

```
class astropy.time.TimeFromEpoch(val1, val2, scale, precision, in_subfmt, out_subfmt,
                                  from_jd=False)
```

Bases: `astropy.time.TimeFormat`

Base class for times that represent the interval from a particular epoch as a floating point multiple of a unit time interval (e.g. seconds or days).

Attributes Summary

<code>value</code>

Methods Summary

<code>set_jds(val1, val2)</code>	Initialize the internal <code>jd1</code> and <code>jd2</code> attributes given <code>val1</code> and <code>val2</code> .
----------------------------------	--

Attributes Documentation

value

Methods Documentation

set_jds (*val1*, *val2*)

Initialize the internal `jd1` and `jd2` attributes given `val1` and `val2`. For an `TimeFromEpoch` subclass like `TimeUnix` these will be floats giving the effective seconds since an epoch time (e.g. 1970-01-01 00:00:00).

TimeGPS

```
class astropy.time.TimeGPS(val1, val2, scale, precision, in_subfmt, out_subfmt, from_jd=False)
```

Bases: `astropy.time.TimeFromEpoch`

GPS time: seconds from 1980-01-06 00:00:00 UTC For example, 630720013.0 is midnight on January 1, 2000.

Notes

This implementation is strictly a representation of the number of seconds (including leap seconds) since midnight UTC on 1980-01-06. GPS can also be considered as a time scale which is ahead of TAI by a fixed offset (to within about 100 nanoseconds).

For details, see <http://tycho.usno.navy.mil/gpstt.html>

Attributes Summary

<code>epoch_format</code>	<code>unicode(string [, encoding[, errors]]) -> object</code>
---------------------------	--

Continued on next page

Table 9.23 – continued from previous page

<code>epoch_scale</code>	<code>unicode(string [, encoding[, errors]]) -> object</code>
<code>epoch_val</code>	<code>unicode(string [, encoding[, errors]]) -> object</code>
<code>epoch_val2</code>	
<code>name</code>	<code>unicode(string [, encoding[, errors]]) -> object</code>
<code>unit</code>	<code>float(x) -> floating point number</code>

Attributes Documentation`epoch_format = u'iso'``epoch_scale = u'tai'``epoch_val = u'1980-01-06 00:00:19'``epoch_val2 = None``name = u'gps'``unit = 1.1574074074074073e-05`**TimeISO****class** `astropy.time.TimeISO` (*val1, val2, scale, precision, in_subfmt, out_subfmt, from_jd=False*)Bases: `astropy.time.TimeString`

ISO 8601 compliant date-time format “YYYY-MM-DD HH:MM:SS.sss...”. For example, 2000-01-01 00:00:00.000 is midnight on January 1, 2000.

The allowed subformats are:

- ‘date_hms’: date + hours, mins, secs (and optional fractional secs)
- ‘date_hm’: date + hours, mins
- ‘date’: date

Attributes Summary

<code>name</code>	<code>unicode(string [, encoding[, errors]]) -> object</code>
<code>subfmts</code>	<code>tuple() -> empty tuple</code>

Attributes Documentation`name = u'iso'``subfmts = ((u'date_hms', u'%Y-%m-%d %H:%M:%S', u'{year:d}-{mon:02d}-{day:02d} {hour:02d}:{min:02d}:{sec:0`

TimeISOT

class `astropy.time.TimeISOT` (*val1, val2, scale, precision, in_subfmt, out_subfmt, from_jd=False*)

Bases: `astropy.time.TimeString`

ISO 8601 compliant date-time format “YYYY-MM-DDTHH:MM:SS.sss...”. This is the same as TimeISO except for a “T” instead of space between the date and time. For example, 2000-01-01T00:00:00.000 is midnight on January 1, 2000.

The allowed subformats are:

- ‘date_hms’: date + hours, mins, secs (and optional fractional secs)
- ‘date_hm’: date + hours, mins
- ‘date’: date

Attributes Summary

<code>name</code>	<code>unicode(string [, encoding[, errors]]) -> object</code>
<code>subfmts</code>	<code>tuple() -> empty tuple</code>

Attributes Documentation

`name = u'isot'`

`subfmts = ((u'date_hms', u'%Y-%m-%dT%H:%M:%S', u'{year:d}-{mon:02d}-{day:02d}T{hour:02d}:{min:02d}:{sec:02d}')`

TimeJD

class `astropy.time.TimeJD` (*val1, val2, scale, precision, in_subfmt, out_subfmt, from_jd=False*)

Bases: `astropy.time.TimeFormat`

Julian Date time format. This represents the number of days since the beginning of the Julian Period. For example, 2451544.5 in JD is midnight on January 1, 2000.

Attributes Summary

<code>name</code>	<code>unicode(string [, encoding[, errors]]) -> object</code>
<code>value</code>	

Methods Summary

`set_jds(val1, val2)`

Attributes Documentation**name = u'jd'****value****Methods Documentation****set_jds** (*val1*, *val2*)**TimeJulianEpoch**

```
class astropy.time.TimeJulianEpoch (val1, val2, scale, precision, in_subfmt, out_subfmt,
                                     from_jd=False)
```

Bases: `astropy.time.TimeEpochDate`

Julian Epoch year as floating point value(s) like 2000.0

Attributes Summary

<code>epoch_to_jd</code>	unicode(string [, encoding[, errors]]) -> object
<code>jd_to_epoch</code>	unicode(string [, encoding[, errors]]) -> object
<code>name</code>	unicode(string [, encoding[, errors]]) -> object
<code>unit</code>	float(x) -> floating point number

Attributes Documentation**epoch_to_jd = u'julian_epoch_jd'****jd_to_epoch = u'jd_julian_epoch'****name = u'jyear'****unit = 365.25****TimeJulianEpochString**

```
class astropy.time.TimeJulianEpochString (val1, val2, scale, precision, in_subfmt, out_subfmt,
                                             from_jd=False)
```

Bases: `astropy.time.TimeEpochDateString`

Julian Epoch year as string value(s) like 'J2000.0'

Attributes Summary

<code>epoch_prefix</code>	<code>unicode(string [, encoding[, errors]]) -> object</code>
<code>epoch_to_jd</code>	<code>unicode(string [, encoding[, errors]]) -> object</code>
<code>jd_to_epoch</code>	<code>unicode(string [, encoding[, errors]]) -> object</code>
<code>name</code>	<code>unicode(string [, encoding[, errors]]) -> object</code>

Attributes Documentation

`epoch_prefix = u'J'`

`epoch_to_jd = u'julian_epoch_jd'`

`jd_to_epoch = u'jd_julian_epoch'`

`name = u'jyear_str'`

TimeMJD

class `astropy.time.TimeMJD` (*val1, val2, scale, precision, in_subfmt, out_subfmt, from_jd=False*)

Bases: `astropy.time.TimeFormat`

Modified Julian Date time format. This represents the number of days since midnight on November 17, 1858. For example, 51544.0 in MJD is midnight on January 1, 2000.

Attributes Summary

<code>name</code>	<code>unicode(string [, encoding[, errors]]) -> object</code>
<code>value</code>	

Methods Summary

`set_jds(val1, val2)`

Attributes Documentation

`name = u'mjd'`

`value`

Methods Documentation

`set_jds` (*val1*, *val2*)

TimePlotDate

`class astropy.time.TimePlotDate` (*val1*, *val2*, *scale*, *precision*, *in_subfmt*, *out_subfmt*,
from_jd=False)

Bases: `astropy.time.TimeFromEpoch`

Matplotlib `plot_date` input: 1 + number of days from 0001-01-01 00:00:00 UTC

This can be used directly in the matplotlib `plot_date` function:

```
>>> import matplotlib.pyplot as plt
>>> jyear = np.linspace(2000, 2001, 20)
>>> t = Time(jyear, format='jyear', scale='utc')
>>> plt.plot_date(t.plot_date, jyear)
>>> plt.gcf().autofmt_xdate() # orient date labels at a slant
>>> plt.draw()
```

For example, 730120.0003703703 is midnight on January 1, 2000.

Attributes Summary

<code>epoch_format</code>	unicode(string [, encoding[, errors]]) -> object
<code>epoch_scale</code>	unicode(string [, encoding[, errors]]) -> object
<code>epoch_val</code>	float(x) -> floating point number
<code>epoch_val2</code>	
<code>name</code>	unicode(string [, encoding[, errors]]) -> object
<code>unit</code>	float(x) -> floating point number

Attributes Documentation

`epoch_format = u'jd'`

`epoch_scale = u'utc'`

`epoch_val = 1721424.5`

`epoch_val2 = None`

`name = u'plot_date'`

`unit = 1.0`

TimeString

class `astropy.time.TimeString` (*val1, val2, scale, precision, in_subfmt, out_subfmt, from_jd=False*)

Bases: `astropy.time.core.TimeUnique`

Base class for string-like time representations.

This class assumes that anything following the last decimal point to the right is a fraction of a second.

This is a reference implementation can be made much faster with effort.

Attributes Summary

<code>value</code>

Methods Summary

<code>set_jds(val1, val2)</code>	Parse the time strings contained in <code>val1</code> and set <code>jd1, jd2</code>
<code>str_kwargs()</code>	Generator that yields a dict of values corresponding to the calendar date and time for the internal JD values.

Attributes Documentation

value

Methods Documentation

set_jds (*val1, val2*)

Parse the time strings contained in `val1` and set `jd1, jd2`

str_kwargs ()

Generator that yields a dict of values corresponding to the calendar date and time for the internal JD values.

TimeUnix

class `astropy.time.TimeUnix` (*val1, val2, scale, precision, in_subfmt, out_subfmt, from_jd=False*)

Bases: `astropy.time.TimeFromEpoch`

Unix time: seconds from 1970-01-01 00:00:00 UTC. For example, 946684800.0 in Unix time is midnight on January 1, 2000.

NOTE: this quantity is not exactly unix time and differs from the strict POSIX definition by up to 1 second on days with a leap second. POSIX unix time actually jumps backward by 1 second at midnight on leap second days while this class value is monotonically increasing at 86400 seconds per UTC day.

Attributes Summary

<code>epoch_format</code>	<code>unicode(string [, encoding[, errors]]) -> object</code>
---------------------------	--

Continued on next page

Table 9.35 – continued from previous page

<code>epoch_scale</code>	<code>unicode(string [, encoding[, errors]]) -> object</code>
<code>epoch_val</code>	<code>unicode(string [, encoding[, errors]]) -> object</code>
<code>epoch_val2</code>	
<code>name</code>	<code>unicode(string [, encoding[, errors]]) -> object</code>
<code>unit</code>	<code>float(x) -> floating point number</code>

Attributes Documentation`epoch_format = u'iso'``epoch_scale = u'utc'``epoch_val = u'1970-01-01 00:00:00'``epoch_val2 = None``name = u'unix'``unit = 1.1574074074074073e-05`**TimeYearDayTime**

`class astropy.time.TimeYearDayTime` (*val1*, *val2*, *scale*, *precision*, *in_subfmt*, *out_subfmt*,
from_jd=False)

Bases: `astropy.time.TimeString`

Year, day-of-year and time as “YYYY:DOY:HH:MM:SS.sss...”. The day-of-year (DOY) goes from 001 to 365 (366 in leap years). For example, 2000:001:00:00:00.000 is midnight on January 1, 2000.

The allowed subformats are:

- ‘date_hms’: date + hours, mins, secs (and optional fractional secs)
- ‘date_hm’: date + hours, mins
- ‘date’: date

Attributes Summary

<code>name</code>	<code>unicode(string [, encoding[, errors]]) -> object</code>
<code>subfmts</code>	<code>tuple() -> empty tuple</code>

Attributes Documentation`name = u'yday'`
`subfmts = ((u'date_hms', u'%Y:%j:%H:%M:%S', u'{year:d}:{yday:03d}:{hour:02d}:{min:02d}:{sec:02d}'), (u'date_h`

Class Inheritance Diagram

9.5 Acknowledgments and Licenses

This package makes use of the [ERFA Software](#) ANSI C library. The copyright of the ERFA software belongs to the NumFOCUS Foundation. The library is made available under the terms of the “BSD-three clauses” license.

The ERFA library is derived, with permission, from the International Astronomical Union’s “Standards of Fundamental Astronomy” library, available from <http://www.iausofa.org>.

ASTRONOMICAL COORDINATE SYSTEMS (ASTROPY.COORDINATES)

10.1 Introduction

The `coordinates` package provides classes for representing a variety of celestial/spatial coordinates, as well as tools for converting between common coordinate systems in a uniform way.

Note: If you have existing code that uses `coordinates` functionality from Astropy version 0.3.x or earlier, please see the section on [Migrating from pre-v0.4 coordinates](#). The interface has changed in ways that are not backward compatible in many circumstances.

10.2 Getting Started

The simplest way to use `coordinates` is to use the `SkyCoord` class. `SkyCoord` objects are instantiated with a flexible and natural approach that supports inputs provided in a number of convenient formats. The following ways of initializing a coordinate are all equivalent:

```
>>> from astropy import units as u
>>> from astropy.coordinates import SkyCoord

>>> c = SkyCoord(ra=10.5*u.degree, dec=41.2*u.degree, frame='icrs')
>>> c = SkyCoord(10.5, 41.2, 'icrs', unit='deg')
>>> c = SkyCoord('00h42m00s', '+41d12m00s', 'icrs')
>>> c = SkyCoord('00 42 00 +41 12 00', 'icrs', unit=(u.hourangle, u.deg))
>>> c
<SkyCoord (ICRS): ra=10.5 deg, dec=41.2 deg>
```

The examples above illustrate a few simple rules to follow when creating a coordinate object:

- Coordinate values can be provided either as unnamed positional arguments or via keyword arguments like `ra`, `dec`, `l`, or `b` (depending on the frame).
- Coordinate frame value is optional and can be specified as a positional argument or via the `frame` keyword.
- Angle units must be specified, either in the values themselves (e.g. `10.5*u.degree` or `'+41d12m00s'`) or via the `unit` keyword.

The individual components of equatorial coordinates are `Longitude` or `Latitude` objects, which are specialized versions of the general `Angle` class. The component values are accessed using aptly named attributes:

```
>>> c = SkyCoord(ra=10.68458*u.degree, dec=41.26917*u.degree)
>>> c.ra
<Longitude 10.68458 deg>
```

```
>>> c.ra.hour
0.7123053333333335
>>> c.ra.hms
hms_tuple(h=0.0, m=42.0, s=44.299200000000525)
>>> c.dec
<Latitude 41.26917 deg>
>>> c.dec.degree
41.26917
>>> c.dec.radian
0.7202828960652683
```

Coordinates can easily be converted to strings using the `to_string()` method:

```
>>> c = SkyCoord(ra=10.68458*u.degree, dec=41.26917*u.degree)
>>> c.to_string('decimal')
'10.6846 41.2692'
>>> c.to_string('dms')
'10d41m04.488s 41d16m09.012s'
>>> c.to_string('hmsdms')
'00h42m44.2992s +41d16m09.012s'
```

For more control over the string formatting, use the `to_string` method of the individual components:

```
>>> c.ra.to_string(decimal=True)
'10.6846'
>>> c.dec.to_string(format='latex')
'$41^\circ16\prime09.012\prime$'
>>> msg = 'My coordinates are: ra="{0}" dec="{1}"'
>>> msg.format(c.ra.to_string(sep=':'), c.dec.to_string(sep=':'))
'My coordinates are: ra="10:41:04.488" dec="41:16:09.012"'
```

Many of the above examples did not explicitly specify the coordinate frame. This is fine if you do not need to transform to other frames or compare with coordinates defined in a different frame. However, to use the full power of `coordinates`, you should specify the reference frame your coordinates are defined in:

```
>>> c_icrs = SkyCoord(ra=10.68458*u.degree, dec=41.26917*u.degree, frame='icrs')
```

Once you've defined the frame of your coordinates, you can transform from that frame to another frame. You can do this a few different ways: if you just want the default version of that frame, you can use attribute-style access. For more control, you can use the `transform_to` method, which accepts a frame name, frame class, or frame instance:

```
>>> from astropy.coordinates import FK5
>>> c_icrs.galactic
<SkyCoord (Galactic): l=121.174302631 deg, b=-21.5728000618 deg>
>>> c_fk5 = c_icrs.transform_to('fk5') # c_icrs.fk5 does the same thing
>>> c_fk5
<SkyCoord (FK5): equinox=J2000.000, ra=10.6845915393 deg, dec=41.2691714591 deg>
>>> c_fk5.transform_to(FK5(equinox='J1975')) # precess to a different equinox
<SkyCoord (FK5): equinox=J1975.000, ra=10.3420913461 deg, dec=41.1323211229 deg>
```

`SkyCoord` and all other `coordinates` objects also support array coordinates. These work the same as single-value coordinates, but they store multiple coordinates in a single object. When you're going to apply the same operation to many different coordinates (say, from a catalog), this is a better choice than a list of `SkyCoord` objects, because it will be *much* faster than applying the operation to each `SkyCoord` in a for loop.

```
>>> SkyCoord(ra=[10, 11]*u.degree, dec=[41, -5]*u.degree)
<SkyCoord (ICRS): (ra, dec) in deg
[(10.0, 41.0), (11.0, -5.0)]>
```

So far we have been using a spherical coordinate representation in all the examples, and this is the default for the built-in frames. Frequently it is convenient to initialize or work with a coordinate using a different representation such as cartesian or cylindrical. This can be done by setting the `representation` for either `SkyCoord` objects or low-level frame coordinate objects:

```
>>> c = SkyCoord(x=1, y=2, z=3, unit='kpc', frame='icrs', representation='cartesian')
>>> c
<SkyCoord (ICRS): x=1.0 kpc, y=2.0 kpc, z=3.0 kpc>
>>> c.x, c.y, c.z
(<Quantity 1.0 kpc>, <Quantity 2.0 kpc>, <Quantity 3.0 kpc>)

>>> c.representation = 'cylindrical'
>>> c
<SkyCoord (ICRS): rho=2.2360679775 kpc, phi=63.4349488229 deg, z=3.0 kpc>
>>> c.phi
<Angle 63.434948... deg>
>>> c.phi.to(u.radian)
<Angle 1.107148... rad>

>>> c.representation = 'spherical'
>>> c
<SkyCoord (ICRS): ra=63.4349488229 deg, dec=53.3007747995 deg, distance=3.74165738677 kpc>

>>> c.representation = 'unitspherical'
>>> c
<SkyCoord (ICRS): ra=63.4349488229 deg, dec=53.3007747995 deg>
```

`SkyCoord` defines a number of convenience methods as well, like on-sky separation between two coordinates and catalog matching (detailed in *Matching Catalogs*):

```
>>> c1 = SkyCoord(ra=10*u.degree, dec=9*u.degree, frame='icrs')
>>> c2 = SkyCoord(ra=11*u.degree, dec=10*u.degree, frame='fk5')
>>> c1.separation(c2) # Differing frames handled correctly
<Angle 1.4045335865905868 deg>
```

Distance from the origin (which is system-dependent, but often the Earth center) can also be assigned to a `SkyCoord`. With two angles and a distance, a unique point in 3D space is available, which also allows conversion to the Cartesian representation of this location:

```
>>> from astropy.coordinates import Distance
>>> c = SkyCoord(ra=10.68458*u.degree, dec=41.26917*u.degree, distance=770*u.kpc)
>>> c.cartesian.x
<Quantity 568.7128654235232 kpc>
>>> c.cartesian.y
<Quantity 107.3008974042025 kpc>
>>> c.cartesian.z
<Quantity 507.88994291875713 kpc>
```

With distances assigned, `SkyCoord` convenience methods are more powerful, as they can make use of the 3D information. For example:

```
>>> c1 = SkyCoord(ra=10*u.degree, dec=9*u.degree, distance=10*u.pc, frame='icrs')
>>> c2 = SkyCoord(ra=11*u.degree, dec=10*u.degree, distance=11.5*u.pc, frame='icrs')
>>> c1.separation_3d(c2)
<Distance 1.5228602415117989 pc>
```

Finally, the `astropy.coordinates` subpackage also provides a quick way to get coordinates for named objects assuming you have an active internet connection. The `from_name` method of `SkyCoord` uses `Sesame` to retrieve coordinates for a particular named object:

```
>>> SkyCoord.from_name("M42")
<SkyCoord (ICRS): ra=83.82208 deg, dec=-5.39111 deg>
```

Note: `from_name` is intended to be a convenience, and is rather simple. If you need precise coordinates for an object you should find the appropriate reference for that measurement and input the coordinates manually.

10.3 Overview of `astropy.coordinates` concepts

Note: The `coordinates` package from v0.4 onward builds from previous versions of the package, and more detailed information and justification of the design is available in [APE \(Astropy Proposal for Enhancement\) 5](#).

Here we provide an overview of the package and associated framework. This background information is not necessary for simply using `coordinates`, particularly if you use the `SkyCoord` high-level class, but it is helpful for more advanced usage, particularly creating your own frame, transformations, or representations. Another useful piece of background information are some *Important Definitions* as they are used in `coordinates`.

`coordinates` is built on a three-tiered system of objects: representations, frames, and a high-level class. Representations classes are a particular way of storing a three-dimensional data point (or points), such as Cartesian coordinates or spherical polar coordinates. Frames are particular reference frames like FK5 or ICRS, which may store their data in different representations, but have well-defined transformations between each other. These transformations are all stored in the `astropy.coordinates.frame_transform_graph`, and new transformations can be created by users. Finally, the high-level class (`SkyCoord`) uses the frame classes, but provides a more accessible interface to these objects as well as various convenience methods and more string-parsing capabilities.

Separating these concepts makes it easier to extend the functionality of `coordinates`. It allows representations, frames, and transformations to be defined or extended separately, while still preserving the high-level capabilities and simplicity of the `SkyCoord` class.

10.4 Using `astropy.coordinates`

More detailed information on using the package is provided on separate pages, listed below.

10.4.1 Working with Angles

The angular components of the various coordinate objects are represented by objects of the `Angle` class. While most likely to be encountered in the context of coordinate objects, `Angle` objects can also be used on their own wherever a representation of an angle is needed.

Creation

The creation of an `Angle` object is quite flexible and supports a wide variety of input object types and formats. The type of the input angle(s) can be an array, scalar, tuple, string, `Quantity` or another `Angle`. This is best illustrated with a number of examples of valid ways to create an `Angle`:

```
>>> import numpy as np
>>> from astropy import units as u
>>> from astropy.coordinates import Angle

>>> Angle('10.2345d')           # String with 'd' abbreviation for degrees
```

```

<Angle 10.2345 deg>
>>> Angle(['10.2345d', '-20d']) # Array of strings
<Angle [ 10.2345,-20.    ] deg>
>>> Angle('1:2:30.43 degrees') # Sexagesimal degrees
<Angle 1.04178611111111113 deg>
>>> Angle('1 2 0 hours') # Sexagesimal hours
<Angle 1.0333333333333334 hourangle>
>>> Angle(np.arange(1., 8.), unit=u.deg) # Numpy array from 1..7 in degrees
<Angle [ 1., 2., 3., 4., 5., 6., 7.] deg>
>>> Angle(u'1°2'3"') # Unicode degree, arcmin and arcsec symbols
<Angle 1.0341666666666667 deg>
>>> Angle('1d2m3.4s') # Degree, arcmin, arcsec.
<Angle 1.0342777777777779 deg>
>>> Angle('-1h2m3s') # Hour, minute, second
<Angle -1.0341666666666667 hourangle>
>>> Angle((-1, 2, 3), unit=u.deg) # (degree, arcmin, arcsec)
<Angle -1.0341666666666667 deg>
>>> Angle(10.2345 * u.deg) # From a Quantity object in degrees
<Angle 10.2345 deg>
>>> Angle(Angle(10.2345 * u.deg)) # From another Angle object
<Angle 10.2345 deg>

```

Representation

The `Angle` object also supports a variety of ways of representing the value of the angle, both as a floating point number as a string:

```

>>> a = Angle(1, u.radian)
>>> a
<Angle 1.0 rad>
>>> a.radian
1.0
>>> a.degree
57.29577951308232
>>> a.hour
3.8197186342054885
>>> a.hms
hms_tuple(h=3.0, m=49.0, s=10.987083139758766)
>>> a.dms
dms_tuple(d=57.0, m=17.0, s=44.806247096362313)
>>> a.signed_dms
signed_dms_tuple(sign=1.0, d=57.0, m=17.0, s=44.806247096362313)
>>> (-a).dms
dms_tuple(d=-57.0, m=-17.0, s=-44.806247096362313)
>>> (-a).signed_dms
signed_dms_tuple(sign=-1.0, d=57.0, m=17.0, s=44.806247096362313)
>>> a.arcminute
3437.7467707849396
>>> a.to_string()
u'1rad'
>>> a.to_string(unit=u.degree)
u'57d17m44.8062s'
>>> a.to_string(unit=u.degree, sep=':')
u'57:17:44.8062'
>>> a.to_string(unit=u.degree, sep=('deg', 'm', 's'))
u'57deg17m44.8062s'
>>> a.to_string(unit=u.hour)

```

```
u'3h49m10.9871s'  
>>> a.to_string(unit=u.hour, decimal=True)  
u'3.81972'
```

Usage

Angles will also behave correctly for appropriate arithmetic operations:

```
>>> a = Angle(1.0, u.radian)  
>>> a + 0.5 * u.radian + 2 * a  
<Angle 3.5 rad>  
>>> np.sin(a / 2)  
<Quantity 0.479425538604203>  
>>> a == a  
array(True, dtype=bool)  
>>> a == (a + a)  
array(False, dtype=bool)
```

`Angle` objects can also be used for creating coordinate objects:

```
>>> from astropy.coordinates import ICRS  
>>> ICRS(Angle(1, u.radian), Angle(0.5, u.radian))  
<ICRS Coordinate: ra=57.2957795131 deg, dec=28.6478897565 deg>
```

Wrapping and bounds

There are two utility methods that simplify working with angles that should have bounds. The `wrap_at()` method allows taking an angle or angles and wrapping to be within a single 360 degree slice. The `is_within_bounds()` method returns a boolean indicating whether an angle or angles is within the specified bounds.

Longitude and Latitude objects

`Longitude` and `Latitude` are two specialized subclasses of the `Angle` class that are used for all of the spherical coordinate classes. `Longitude` is used to represent values like right ascension, Galactic longitude, and azimuth (for ecliptic, Galactic, and Alt-Az coordinates, respectively). `Latitude` is used for declination, Galactic latitude, and elevation.

Longitude

A `Longitude` object is distinguished from a pure `Angle` by virtue of a `wrap_angle` property. The `wrap_angle` specifies that all angle values represented by the object will be in the range:

```
wrap_angle - 360 * u.deg <= angle(s) < wrap_angle
```

The default `wrap_angle` is 360 deg. Setting '`wrap_angle=180 * u.deg`' would instead result in values between -180 and +180 deg. Setting the `wrap_angle` attribute of an existing `Longitude` object will result in re-wrapping the angle values in-place. For example:

```
>>> from astropy.coordinates import Longitude  
>>> a = Longitude([-20, 150, 350, 360] * u.deg)  
>>> a.degree  
array([ 340., 150., 350.,  0.])  
>>> a.wrap_angle = 180 * u.deg
```

```
>>> a.degree
array([-20., 150., -10.,  0.]
```

Latitude

A `Latitude` object is distinguished from a pure `Angle` by virtue of being bounded so that:

```
-90.0 * u.deg <= angle(s) <= +90.0 * u.deg
```

Any attempt to set a value outside that range will result in a `ValueError`.

10.4.2 Using the SkyCoord High-level Class

The `SkyCoord` class provides a simple and flexible user interface for celestial coordinate representation, manipulation, and transformation between coordinate frames. This is a high-level class that serves as a wrapper around the low-level coordinate frame classes like `ICRS` and `FK5` which do most of the heavy lifting.

The key distinctions between `SkyCoord` and the low-level classes (*Using and Designing Coordinate Frames*) are as follows:

- The `SkyCoord` object can maintain the union of frame attributes for all built-in and user-defined coordinate frames in the `astropy.coordinates.frame_transform_graph`. Individual frame classes hold only the required attributes (e.g. equinox, observation time or observer location) for that frame. This means that a transformation from `FK4` (with equinox and observation time) to `ICRS` (with neither) and back to `FK4` via the low-level classes would not remember the original equinox and observation time. Since the `SkyCoord` object stores all attributes, such a round-trip transformation will return to the same coordinate object.
- The `SkyCoord` class is more flexible with inputs to accommodate a wide variety of user preferences and available data formats.
- The `SkyCoord` class has a number of convenience methods that are useful in typical analysis.
- At present, `SkyCoord` objects can use only coordinate frames that have transformations defined in the `astropy.coordinates.frame_transform_graph` transform graph object.

Creating SkyCoord objects

The `SkyCoord` class accepts a wide variety of inputs for initialization. At a minimum these must provide one or more celestial coordinate values with unambiguous units. Typically one also specifies the coordinate frame, though this is not required.

Common patterns are shown below. In this description the values in upper case like `COORD` or `FRAME` represent inputs which are described in detail in the [Initialization Syntax](#) section. Elements in square brackets like `[unit=UNIT]` are optional.

```
SkyCoord(COORD, [FRAME], keyword_args ...)
SkyCoord(LON, LAT, [frame=FRAME], [unit=UNIT], keyword_args ...)
SkyCoord([FRAME], <lon_attr>=LON, <lat_attr>=LAT, keyword_args ...)
```

The examples below illustrate common ways of initializing a `SkyCoord` object. These all reflect initializing using spherical coordinates, which is the default for all built-in frames. In order to understand working with coordinates using a different representation such as cartesian or cylindrical, see the section on [Representations](#). First some imports:

```
>>> from astropy.coordinates import SkyCoord # High-level coordinates
>>> from astropy.coordinates import ICRS, Galactic, FK4, FK5 # Low-level frames
>>> from astropy.coordinates import Angle, Latitude, Longitude # Angles
>>> import astropy.units as u
>>> import numpy as np
```

The coordinate values and frame specification can now be provided using positional and keyword arguments. First we show positional arguments for RA and Dec:

```
>>> SkyCoord(10, 20, unit="deg") # Defaults to ICRS
<SkyCoord (ICRS): ra=10.0 deg, dec=20.0 deg>

>>> SkyCoord([1, 2, 3], [-30, 45, 8], "icrs", unit="deg")
<SkyCoord (ICRS): (ra, dec) in deg
  [(1.0, -30.0), (2.0, 45.0), (3.0, 8.0)]>
```

Notice that the first example above does not explicitly give a frame. In this case, the default is taken to be the ICRS system (approximately correct for “J2000” equatorial coordinates). It is always better to explicitly specify the frame when it is known to be ICRS, however, as anyone reading the code will be better able to understand the intent.

String inputs in common formats are acceptable, and the frame can be supplied as either a class type like `FK4` or the lower-case version of the name as a string, e.g. “fk4”:

```
>>> coords = ["1:12:43.2 +1:12:43", "1 12 43.2 +1 12 43"]
>>> sc = SkyCoord(coords, FK4, unit=(u.hourangle, u.deg), obstime="J1992.21")
>>> sc = SkyCoord(coords, 'fk4', unit='hourangle,deg', obstime="J1992.21")

>>> sc = SkyCoord("1h12m43.2s", "+1d12m43s", Galactic) # Units from strings
>>> sc = SkyCoord("1h12m43.2s +1d12m43s", Galactic) # Units from string
>>> sc = SkyCoord("galactic", l="1h12m43.2s", b="+1d12m43s")
```

Astropy `Quantity`-type objects are acceptable and encouraged as a form of input:

```
>>> ra = Longitude([1, 2, 3], unit=u.deg) # Could also use Angle
>>> dec = np.array([4.5, 5.2, 6.3]) * u.deg # Astropy Quantity
>>> sc = SkyCoord(ra, dec, frame='icrs')
>>> sc = SkyCoord(ICRS, ra=ra, dec=dec, obstime='2001-01-02T12:34:56')
```

Finally it is possible to initialize from a low-level coordinate frame object.

```
>>> c = FK4(1 * u.deg, 2 * u.deg)
>>> sc = SkyCoord(c, obstime='J2010.11', equinox='B1965') # Override defaults
```

A key subtlety highlighted here is that when low-level objects are created they have certain default attribute values. For instance the `FK4` frame uses `equinox='B1950.0'` and `obstime=equinox` as defaults. If this object is used to initialize a `SkyCoord` it is possible to override the low-level object attributes that were not explicitly set. If the coordinate above were created with `c = FK4(1 * u.deg, 2 * u.deg, equinox='B1960')` then creating a `SkyCoord` with a different equinox would raise an exception.

Initialization Syntax

For spherical representations, which are the most common and are the default input format for all built-in frames, the syntax for `SkyCoord` is given below:

```
SkyCoord(COORD, [FRAME | frame=FRAME], [unit=UNIT], keyword_args ...)
SkyCoord(LON, LAT, [DISTANCE], [FRAME | frame=FRAME], [unit=UNIT], keyword_args ...)
SkyCoord([FRAME | frame=FRAME], <lon_name>=LON, <lat_name>=LAT, [unit=UNIT],
         keyword_args ...)
```

In the above description, elements in all capital letters (e.g. `FRAME`) describes a user input of that element type. Elements in square brackets are optional. For non-spherical inputs see the [Representations](#) section.

LON, LAT

Longitude and latitude value can be specified as separate positional arguments. The following options are available for longitude and latitude:

- Single angle value:
 - `Quantity` object
 - Plain numeric value with `unit` keyword specifying the unit
 - Angle string which is formatted for *Creation* of `Longitude` or `Latitude` objects
- List or `Quantity` array or numpy array of angle values
- `Angle`, `Longitude`, or `Latitude` object, which can be scalar or array-valued

DISTANCE

The distance to the object from the frame center can be optionally specified:

- Single distance value:
 - `Quantity` or `Distance` object
 - Plain numeric value for a dimensionless distance
 - Plain numeric value with `unit` keyword specifying the unit
- List or `Quantity` or `Distance` array or numpy array of angle values

COORD

This input form uses a single object to supply coordinate data. For the case of spherical coordinate frames, the coordinate can include one or more longitude and latitude pairs in one of the following ways:

- Single coordinate string with a LON and LAT value separated by a space. The respective values can be any string which is formatted for *Creation* of `Longitude` or `Latitude` objects, respectively.
- List or numpy array of such coordinate strings
- List of (LON, LAT) tuples, where each LON and LAT are scalars (not arrays)
- $N \times 2$ numpy or `Quantity` array of values where the first column is longitude and the second column is latitude, e.g. `[[270, -30], [355, +85]] * u.deg`
- List of (LON, LAT, DISTANCE) tuples
- $N \times 3$ numpy or `Quantity` array of values where columns are longitude, latitude, and distance respectively.

The input can also be more generalized objects that are not necessarily represented in the standard spherical coordinates:

- Coordinate frame object, e.g. `FK4(1*u.deg, 2*u.deg, obstime='J2012.2')`
- `SkyCoord` object (which just makes a copy of the object)
- `BaseRepresentation` subclass object like `SphericalRepresentation`, `CylindricalRepresentation`, or `CartesianRepresentation`.

FRAME

This can be a `BaseCoordinateFrame` frame class or the corresponding string alias. The frame classes that are built in to astropy are `ICRS`, `FK5`, `FK4`, `FK4NoETerms`, `Galactic`, and `AltAz`. The string aliases are simply lower-case versions of the class name.

If the frame is not supplied then you will see a special ICRS identifier. This indicates that the frame is unspecified and operations that require comparing coordinates (even within that object) are not allowed.

unit=UNIT

The unit specifier can be one of the following:

- `Unit` object which is an angular unit that is equivalent to `Unit('radian')`
- Single string with a valid angular unit name
- 2-tuple of `Unit` objects or string unit names specifying the LON and LAT unit respectively, e.g. `('hourangle', 'degree')`
- Single string with two unit names separated by a comma, e.g. `'hourangle, degree'`

If only a single unit is provided then it applies to both LON and LAT.

Other keyword arguments

In lieu of positional arguments to specify the longitude and latitude, the frame-specific names can be used as keyword arguments:

***ra, dec*: LON, LAT values, optional**

RA and Dec for frames where these are representation, including [FIXME] `ICRS`, `FK5`, `FK4`, and `FK4NoETerms`.

***l, b*: LON, LAT values, optional**

Galactic `l` and `b` for the `Galactic` frame.

The following keywords can be specified for any frame:

***distance*: valid `Distance` initializer, optional**

Distance from reference from center to source.

***obstime*: valid `Time` initializer, optional**

Time of observation

***equinox*: valid `Time` initializer, optional**

Coordinate frame equinox

If custom user-defined frames are included in the transform graph and they have additional frame attributes, then those attributes can also be set via corresponding keyword args in the `SkyCoord` initialization.

Array operations

It is possible to store arrays of coordinates in a `SkyCoord` object, and manipulations done in this way will be orders of magnitude faster than looping over a list of individual `SkyCoord` objects:

```
>>> ra = np.random.uniform(0, 360, size=1000) * u.deg
>>> dec = np.random.uniform(-90, 90, size=1000) * u.deg

>>> sc_list = [SkyCoord(r, d, 'icrs') for r, d in zip(ra, dec)]
>>> timeit sc_gal_list = [c.galactic for c in sc_list]
1 loops, best of 3: 7.66 s per loop

>>> sc = SkyCoord(ra, dec, 'icrs')
>>> timeit sc_gal = sc.galactic
100 loops, best of 3: 8.92 ms per loop
```

In addition to vectorized transformations, you can do the usual array slicing, dicing, and selection:

```

>>> north_mask = sc.dec > 0
>>> sc_north = sc[north_mask]
>>> len(sc_north)
504
>>> sc[2:4]
<SkyCoord (ICRS): (ra, dec) in deg
      [(304.304015..., 6.900282...),
       (322.560148..., 34.872244...)]>
>>> sc[2]
<SkyCoord (ICRS): ra=304.304015... deg, dec=6.900282... deg>

```

Attributes

The `SkyCoord` object has a number of useful attributes which come in handy. By digging through these we'll learn a little bit about `SkyCoord` and how it works.

To begin (if you don't know already) one of the most important tools for learning about attributes and methods of objects is "TAB-discovery". From within IPython you can type an object name, the period, and then the <TAB> key to see what's available. This can often be faster than reading the documentation:

```

>>> sc = SkyCoord(1, 2, 'icrs', unit='deg', obstime='2013-01-02 14:25:36')
>>> sc.<TAB>
sc.cartesian                sc.match_to_catalog_3d
sc.data                    sc.match_to_catalog_sky
sc.dec                     sc.name
sc.default_representation  sc.obstime
sc.distance                sc.position_angle
sc.equinox                 sc.ra
sc.fk4                    sc.realize_frame
sc.fk4noeterms            sc.represent_as
sc.fk5                    sc.representation
sc.frame                   sc.representation_component_names
sc.frame_attr_names       sc.representation_component_units
sc.frame_specific_representation_info
sc.from_name              sc.representation_info
sc.galactic                sc.separation
sc.get_frame_attr_names   sc.separation_3d
sc.has_data                sc.shape
sc.icrs                   sc.spherical
sc.is_frame_attr_default  sc.time_attr_names
sc.is_transformable_to    sc.to_string
sc.isscalar                sc.transform_to

```

Here we see a bunch of stuff there but much of it should be recognizable or easily guessed. The most obvious may be the longitude and latitude attributes which are named `ra` and `dec` for the ICRS frame:

```

>>> sc.ra
<Longitude 1.0 deg>
>>> sc.dec
<Latitude 2.0 deg>

```

Next notice that all the built-in frame names `icrs`, `galactic`, `fk5`, `fk4`, and `fk4noeterms` are there. Through the magic of Python properties, accessing these attributes calls the object `transform_to` method appropriately and returns a new `SkyCoord` object in the requested frame:

```

>>> sc_gal = sc.galactic
>>> sc_gal
<SkyCoord (Galactic): l=99.6379436471 deg, b=-58.7096055983 deg>

```

Other attributes you should recognize are `distance`, `equinox`, `obstime`, `shape`.

Digger deeper

[Casual users can skip this section]

After transforming to Galactic the longitude and latitude values are now labeled `l` and `b`, following the normal convention for Galactic coordinates. How does the object know what to call its values? The answer lies in some less-obvious attributes:

```
>>> sc_gal.representation_component_names
OrderedDict([(u'l', u'lon'), (u'b', u'lat'), (u'distance', u'distance')])

>>> sc_gal.representation_component_units
OrderedDict([(u'l', Unit("deg")), (u'b', Unit("deg"))])

>>> sc_gal.representation
<class 'astropy.coordinates.representation.SphericalRepresentation'>
```

Together these tell the object that `l` and `b` are the longitude and latitude, and that they should both be displayed in units of degrees as a spherical-type coordinate (and not, e.g. a cartesian coordinate). Furthermore the frame's `representation_component_names` attribute defines the coordinate keyword arguments that `SkyCoord` will accept.

Another important attribute is `frame_attr_names`, which defines the additional attributes that are required to fully define the frame:

```
>>> sc_fk4 = SkyCoord(1, 2, 'fk4', unit='deg')
>>> sc_fk4.get_frame_attr_names()
{'equinox': <Time object: scale='tai' format='byear_str' value=B1950.000>,
 u'obstime': None}
```

The key values correspond to the defaults if no explicit value is provide by the user. This example shows that the `FK4` frame has two attributes `equinox` and `obstime` that are required to fully define the frame.

Some trickery is happening here because many of these attributes are actually owned by the underlying coordinate frame object which does much of the real work. This is the middle layer in the three-tiered system of objects: representation (spherical, cartesian, etc.), frame (aka low-level frame class), and `SkyCoord` (aka high-level class):

```
>>> sc.frame
<ICRS Coordinate: ra=1.0 deg, dec=2.0 deg>

>>> sc.has_data is sc.frame.has_data
True

>>> sc.frame.<TAB>
sc.frame.cartesian          sc.frame.ra
sc.frame.data               sc.frame.realize_frame
sc.frame.dec                sc.frame.represent_as
sc.frame.default_representation sc.frame.representation
sc.frame.distance           sc.frame.representation_component_names
sc.frame.frame_attr_names  sc.frame.representation_component_units
sc.frame.frame_specific_representation_info sc.frame.representation_info
sc.frame.get_frame_attr_names sc.frame.separation
sc.frame.has_data           sc.frame.separation_3d
sc.frame.is_frame_attr_default sc.frame.shape
sc.frame.is_transformable_to sc.frame.spherical
sc.frame.isscalar           sc.frame.time_attr_names
```

```
sc.frame.name                sc.frame.transform_to
>>> sc.frame.name
'icrs'
```

The `SkyCoord` object exposes the `frame` object attributes as its own. Though it might seem a tad confusing at first, this a good thing because it makes `SkyCoord` objects and `BaseCoordinateFrame` objects behave very similarly and most routines can accept either one as input without much bother (duck typing!).

The lowest layer in the stack is the abstract `UnitSphericalRepresentation` object:

```
>>> sc_gal.frame.data
<UnitSphericalRepresentation lon=1.739010... rad, lat=-1.024675... rad>
```

Transformations

The topic of transformations is covered in detail in the section on *Transforming Between Systems*.

For completeness here we will give some simple examples. Once you've defined your coordinates and the reference frame, you can transform from that frame to another frame. You can do this a few different ways: if you just want the default version of that frame, you can use attribute-style access (as mentioned previously). For more control, you can use the `transform_to` method, which accepts a frame name, frame class, frame instance, or `SkyCoord`:

```
>>> from astropy.coordinates import FK5
>>> sc = SkyCoord(1, 2, 'icrs', unit='deg')
>>> sc.galactic
<SkyCoord (Galactic): l=99.6379436471 deg, b=-58.7096055983 deg>

>>> sc.transform_to('fk5') # Same as sc.fk5 and sc.transform_to(FK5)
<SkyCoord (FK5): equinox=J2000.000, ra=1.00000655566 deg, dec=2.00000243092 deg>

>>> sc.transform_to(FK5(equinox='J1975')) # Transform to FK5 with a different equinox
<SkyCoord (FK5): equinox=J1975.000, ra=0.679672818323 deg, dec=1.86083014099 deg>
```

Transforming to a `SkyCoord` instance is an easy way of ensuring that two coordinates are in the exact same reference frame:

```
>>> sc2 = SkyCoord(3, 4, 'fk4', unit='deg', obstime='J1978.123', equinox='B1960.0')
>>> sc.transform_to(sc2)
<SkyCoord (FK4): equinox=B1960.000, obstime=J1978.123, ra=0.48726331438 deg, dec=1.77731617297 deg>
```

Representations

So far we have been using a spherical coordinate representation in the all the examples, and this is the default for the built-in frames. Frequently it is convenient to initialize or work with a coordinate using a different representation such as cartesian or cylindrical. In this section we discuss how to initialize an object using a different representation and how to change the representation of an object. For more information about representation objects themselves see *Using and Designing Coordinate Representations*.

Initialization

Most of what you need to know can be inferred from the examples below and by extrapolating the previous documentation for spherical representations. Initialization just requires setting the `representation` keyword and supplying the corresponding components for that representation:

```
>>> c = SkyCoord(x=1, y=2, z=3, unit='kpc', representation='cartesian')
>>> c
<SkyCoord (ICRS): x=1.0 kpc, y=2.0 kpc, z=3.0 kpc>
>>> c.x, c.y, c.z
(<Quantity 1.0 kpc>, <Quantity 2.0 kpc>, <Quantity 3.0 kpc>)
```

Other variations include:

```
>>> SkyCoord(1, 2*u.deg, 3, representation='cylindrical')
<SkyCoord (ICRS): rho=1.0 , phi=2.0 deg, z=3.0 >

>>> SkyCoord(rho=1*u.km, phi=2*u.deg, z=3*u.m, representation='cylindrical')
<SkyCoord (ICRS): rho=1.0 km, phi=2.0 deg, z=3.0 m>

>>> SkyCoord(rho=1, phi=2, z=3, unit=(u.km, u.deg, u.m), representation='cylindrical')
<SkyCoord (ICRS): rho=1.0 km, phi=2.0 deg, z=3.0 m>

>>> SkyCoord(1, 2, 3, unit=(None, u.deg, None), representation='cylindrical')
<SkyCoord (ICRS): rho=1.0 , phi=2.0 deg, z=3.0 >
```

In general terms, the allowed syntax is as follows:

```
SkyCoord(COORD, [FRAME | frame=FRAME], [unit=UNIT], [representation=REPRESENTATION],
         keyword_args ...)
SkyCoord(COMP1, COMP2, [COMP3], [FRAME | frame=FRAME], [unit=UNIT],
         [representation=REPRESENTATION], keyword_args ...)
SkyCoord([FRAME | frame=FRAME], <comp1_name>=COMP1, <comp2_name>=COMP2,
         <comp3_name>=COMP3, [representation=REPRESENTATION], [unit=UNIT],
         keyword_args ...)
```

In this case the `keyword_args` now includes the element `representation=REPRESENTATION`. In the above description, elements in all capital letters (e.g. `FRAME`) describes a user input of that element type. Elements in square brackets are optional.

COMP1, COMP2, COMP3

Component values can be specified as separate positional arguments or as keyword arguments. In this formalism the exact types of allowed input depend on the details of the representation. In general the following input forms are supported:

- Single value:
 - Component class object
 - Plain numeric value with `unit` keyword specifying the unit
- List or component class array or numpy array of values

Each representation component has a specified class (the “component class”) which is used to convert generic input data into a pre-defined object class with a certain unit. These component classes are expected to be subclasses of the `Quantity` class.

COORD

This input form uses a single object to supply coordinate data. The coordinate can specify one or more coordinate positions as follows:

- List of `(COMP1, .., COMP<M>)` tuples, where each component is a scalar (not array) and there are `M` components in the representation. Typically there are 3 components, but some (e.g. `UnitSphericalRepresentation`) can have fewer.

- $N \times M$ numpy or `Quantity` array of values, where N is the number of coordinates and M is the number of components.

REPRESENTATION

The representation can be supplied either as a `BaseRepresentation` class (e.g. `CartesianRepresentation`) or as a string name which is simply the class name in lower case and without the final representation (e.g. `'cartesian'`).

The rest of the inputs for creating a `SkyCoord` object in the general case are the same as for spherical.

Details

The available set of representations is dynamic and may change depending what representation classes have been defined. The built-in representations are:

Name	Class
spherical	<code>SphericalRepresentation</code>
unitspherical	<code>UnitSphericalRepresentation</code>
physicsspherical	<code>PhysicsSphericalRepresentation</code>
cartesian	<code>CartesianRepresentation</code>
cylindrical	<code>CylindricalRepresentation</code>

Each frame knows about all the available representations, but different frames may use different names for the same components. A common example is that the `Galactic` frame uses `l` and `b` instead of `ra` and `dec` for the `lon` and `lat` components of the `SphericalRepresentation`.

For a particular frame, in order to see the full list of representations and how it names all the components, first make an instance of that frame without any data, and then print the `representation_info` property:

```
>>> ICRS().representation_info
{astropy.coordinates.representation.CartesianRepresentation:
  {u'names': (u'x', u'y', u'z'),
   u'units': (None, None, None)},
 astropy.coordinates.representation.SphericalRepresentation:
  {u'names': (u'ra', u'dec', u'distance'),
   u'units': (Unit("deg"), Unit("deg"), None)},
 astropy.coordinates.representation.UnitSphericalRepresentation:
  {u'names': (u'ra', u'dec'),
   u'units': (Unit("deg"), Unit("deg"))},
 astropy.coordinates.representation.PhysicsSphericalRepresentation:
  {u'names': (u'phi', u'theta', u'r'),
   u'units': (Unit("deg"), Unit("deg"), None)},
 astropy.coordinates.representation.CylindricalRepresentation:
  {u'names': (u'rho', u'phi', u'z'),
   u'units': (None, Unit("deg"), None)}
}
```

This is a bit messy but it shows that for each representation there is a `dict` with two keys:

- `names`: defines how each component is named in that frame
- `units`: defines the units of each component when output, where `None` means to not force a particular unit.

For a particular coordinate instance you can use the `representation` attribute in conjunction with the `representation_component_names` attribute to figure out what keywords are accepted by a particular class object. The former will be the representation class the system is expressed in (e.g., `spherical` for equatorial frames), and the latter will be a dictionary mapping names for that frame to the component name on the representation class:

```
>>> import astropy.units as u
>>> icrs = ICRS(1*u.deg, 2*u.deg)
>>> icrs.representation
<class 'astropy.coordinates.representation.SphericalRepresentation'>
>>> icrs.representation_component_names
OrderedDict([(u'ra', u'lon'), (u'dec', u'lat'), (u'distance', u'distance')])
```

Changing representation

The representation of the coordinate object can be changed, as shown below. This actually does *nothing* to the object internal data which stores the coordinate values, but it changes the external view of that data in two ways:

- The object prints itself in accord with the new representation.
- The available attributes change to match those of the new representation (e.g. from `ra`, `dec`, `distance` to `x`, `y`, `z`).

Setting the `representation` thus changes a *property* of the object (how it appears) without changing the intrinsic object itself which represents a point in 3d space.

```
>>> c = SkyCoord(x=1, y=2, z=3, unit='kpc', representation='cartesian')
>>> c
<SkyCoord (ICRS): x=1.0 kpc, y=2.0 kpc, z=3.0 kpc>

>>> c.representation = 'cylindrical'
>>> c
<SkyCoord (ICRS): rho=2.2360679775 kpc, phi=63.4349488229 deg, z=3.0 kpc>
>>> c.phi.to(u.deg)
<Angle 63.43494882292201 deg>
>>> c.x
...
AttributeError: 'SkyCoord' object has no attribute 'x'

>>> c.representation = 'spherical'
>>> c
<SkyCoord (ICRS): ra=63.4349488229 deg, dec=53.3007747995 deg, distance=3.74165738677 kpc>

>>> c.representation = 'unitspherical'
>>> c
<SkyCoord (ICRS): ra=63.4349488229 deg, dec=53.3007747995 deg>
```

You can also use any representation class to set the representation:

```
>>> from astropy.coordinates import CartesianRepresentation
>>> c.representation = CartesianRepresentation
```

Note that if all you want is a particular representation without changing the state of the `SkyCoord` object, you should instead use the `astropy.coordinates.SkyCoord.represent_as()` method:

```
>>> c.representation = 'spherical'
>>> cart = c.represent_as(CartesianRepresentation)
>>> cart
<CartesianRepresentation x=1.0 kpc, y=2.0 kpc, z=3.0 kpc>
>>> c.representation
<class 'astropy.coordinates.representation.SphericalRepresentation'>
```

Convenience methods

A number of convenience methods are available, and you are encouraged to read the available docstrings below:

- `match_to_catalog_sky`,
- `match_to_catalog_3d`,
- `position_angle`,
- `separation`,
- `separation_3d`

Addition information and examples can be found in the section on *Separations, Catalog Matching, and Related Functionality*.

10.4.3 Transforming Between Systems

`astropy.coordinates` supports a rich system for transforming coordinates from one system to another. While common astronomy frames are built into Astropy, the transformation infrastructure is dynamic. This means it allows users to define new coordinate frames and their transformations. The topic of writing your own coordinate frame or transforms is detailed in *Defining a New Frame*, and this section is focused on how to *use* transformations.

The simplest method of transformation is shown below:

```
>>> import astropy.units as u
>>> from astropy.coordinates import SkyCoord
>>> gc = SkyCoord(l=0*u.degree, b=45*u.degree, frame='galactic')
>>> gc.fk5
<SkyCoord (FK5): equinox=J2000.000, ra=229.27250215 deg, dec=-1.12841764184 deg>
```

While this appears to be simple attribute-style access, it is actually syntactic sugar for the more general `transform_to()` method, which can accept either a frame name, class or instance:

```
>>> from astropy.coordinates import FK5
>>> gc.transform_to('fk5')
<SkyCoord (FK5): equinox=J2000.000, ra=229.27250215 deg, dec=-1.12841764184 deg>
>>> gc.transform_to(FK5)
<SkyCoord (FK5): equinox=J2000.000, ra=229.27250215 deg, dec=-1.12841764184 deg>
>>> gc.transform_to(FK5(equinox='J1980.0'))
<SkyCoord (FK5): equinox=J1980.000, ra=229.014681064 deg, dec=-1.05557823687 deg>
```

As a convenience it is also possible to use a `SkyCoord` object as the frame in `transform_to()`. This allows easily putting one coordinate object into the frame of another:

```
>>> sc = SkyCoord(ra=1.0, dec=2.0, unit='deg', frame=FK5, equinox='J1980.0')
>>> gc.transform_to(sc)
<SkyCoord (FK5): equinox=J1980.000, ra=229.014681064 deg, dec=-1.05557823687 deg>
```

The table below summarizes the built-in coordinate frames. For details of these frames and the transformations between them see the `astropy.coordinates` API documentation and the `BaseCoordinateFrame` class which forms the basis for all `astropy.coordinates` coordinate frames.

Frame class	Frame name
ICRS	icrs
FK5	fk5
FK4	fk4
FK4NoETerms	fk4noeterms
Galactic	galactic

Additionally, some coordinate frames (including `FK5`, `FK4`, and `FK4NoETerms`) support “self transformations”, meaning the *type* of frame doesn’t change, but the frame attributes do. Any example is precessing a coordinate from one equinox to another in an equatorial system. This is done by passing `transform_to` a frame class with the relevant attributes, as shown below. Note that these systems use a default equinox if you don’t specify one:

```
>>> fk5c = FK5('02h31m49.09s', '+89d15m50.8s')
>>> fk5c.equinox
<Time object: scale='utc' format='jyear_str' value=J2000.000>
>>> fk5c
<SkyCoord (FK5): equinox=J2000.000, ra=37.9545416667 deg, dec=89.2641111111 deg>
>>> fk5_2005 = FK5(equinox='J2005') # String initializes an astropy.time.Time object
>>> fk5c.transform_to(fk5_2005)
<SkyCoord (FK5): equinox=J2005.000, ra=39.3931763878 deg, dec=89.2858442155 deg>
```

You can also specify the equinox when you create a coordinate using an `Time` object:

```
>>> from astropy.time import Time
>>> fk5c = FK5('02h31m49.09s', '+89d15m50.8s',
...           equinox=Time('J1970', scale='utc'))
>>> fk5_2000 = FK5(equinox=Time(2000, format='jyear', scale='utc'))
>>> fk5c.transform_to(fk5_2000)
<SkyCoord (FK5): equinox=2000.0, ra=48.0231710002 deg, dec=89.386724854 deg>
```

The same lower-level frame classes also have a `transform_to()` method that works the same as above, but they do not support attribute-style access. They are also subtly different in that they only use frame attributes present in the initial or final frame, while `SkyCoord` objects use any frame attributes they have for all transformation steps. So `SkyCoord` can always transform from one frame to another and back again without change, while low-level classes may lose information and hence often do not round-trip.

10.4.4 Formatting Coordinate Strings

Getting a string representation of a coordinate is most powerfully approached by treating the components (e.g., RA and Dec) separately. For example:

```
>>> from astropy.coordinates import ICRS
>>> from astropy import units as u
>>> c = ICRS(187.70592*u.degree, 12.39112*u.degree)
>>> str(c.ra) + ' ' + str(c.dec)
'187d42m21.312s 12d23m28.032s'
```

To get better control over the formatting, you can use the angles’ `to_string()` method (see *Working with Angles* for more). For example:

```
>>> rahmsstr = c.ra.to_string(u.hour)
>>> str(rahmsstr)
'12h30m49.4208s'
>>> decdmsstr = c.dec.to_string(u.degree, alwayssign=True)
>>> str(decdmsstr)
'+12d23m28.032s'
>>> rahmsstr + ' ' + decdmsstr
'u'12h30m49.4208s +12d23m28.032s'
```

You can also use python’s `format` string method to create more complex string expressions, such as IAU-style coordinates or even full sentences:

```
>>> 'SDSS J{0}{1}'.format(c.ra.to_string(sep='', precision=2, pad=True), c.dec.to_string(sep='', prec
```

```
>>> 'The galaxy M87, at an RA of {0.ra.deg:.1f} and Dec of {0.dec.deg:.1f} degrees, has an impressive
The galaxy M87, at an RA of 187.7 and Dec of 12.4 degrees, has an impressive jet.'
```

10.4.5 Separations, Catalog Matching, and Related Functionality

`astropy.coordinates` contains commonly-used tools for comparing or matching coordinate objects. Of particular importance are those for determining separations between coordinates and those for matching a coordinate (or coordinates) to a catalog. These are mainly implemented as methods on the coordinate objects.

Separations

The on-sky separation is easily computed with the `astropy.coordinates.BaseCoordinateFrame.separation()` or `astropy.coordinates.SkyCoord.separation()` methods, which computes the great-circle distance (*not* the small-angle approximation):

```
>>> from astropy import units as u
>>> from astropy.coordinates import SkyCoord
>>> c1 = SkyCoord('5h23m34.5s', '-69d45m22s', frame='icrs')
>>> c2 = SkyCoord('0h52m44.8s', '-72d49m43s', frame='fk5')
>>> sep = c1.separation(c2)
>>> sep
<Angle 20.74611447604398 deg>
```

The returned object is an `Angle` instance, so it is straightforward to access the angle in any of several equivalent angular units:

```
>>> sep.radian
0.36208800460262575
>>> sep.hour
1.3830742984029323
>>> sep.arcminute
1244.7668685626388
>>> sep.arcsecond
74686.01211375833
```

Also note that the two input coordinates were not in the same frame - one is automatically converted to match the other, ensuring that even though they are in different frames, the separation is determined consistently. This does mean, however, that a `SkyCoord` without a frame cannot be compared in this manner:

```
>>> c1 = SkyCoord('5h23m34.5s', '-69d45m22s')
>>> c2 = SkyCoord('0h52m44.8s', '-72d49m43s')
>>> sep = c1.separation(c2)
ValueError: Cannot transform to/from this SkyCoord because the frame was not specified at creation.
```

In addition to the on-sky separation described above, `astropy.coordinates.BaseCoordinateFrame.separation_3d()` or `astropy.coordinates.SkyCoord.separation_3d()` methods will determine the 3D distance between two coordinates that have distance defined:

```
>>> from astropy.coordinates import SkyCoord
>>> c1 = SkyCoord('5h23m34.5s', '-69d45m22s', distance=70*u.kpc, frame='icrs')
>>> c2 = SkyCoord('0h52m44.8s', '-72d49m43s', distance=80*u.kpc, frame='icrs')
>>> sep = c1.separation_3d(c2)
>>> sep
<Distance 28.743988157814094 kpc>
```

Matching Catalogs

`coordinates` supports leverages the coordinate framework to make it straightforward to find the closest coordinates in a catalog to a desired set of other coordinates. For example, assuming `ra1/dec1` and `ra2/dec2` are numpy arrays loaded from some file:

```
>>> from astropy.coordinates import SkyCoord
>>> from astropy import units as u
>>> c = SkyCoord(ra=ra1*u.degree, dec=dec1*u.degree)
>>> catalog = SkyCoord(ra=ra2*u.degree, dec=dec2*u.degree)
>>> idx, d2d, d3d = c.match_to_catalog_sky(catalog)
```

You can also find the nearest 3d matches, different from the on-sky separation shown above only when the coordinates were initialized with a distance:

```
>>> c = SkyCoord(ra=ra1*u.degree, dec=dec1*u.degree, distance=distance1*u.kpc)
>>> catalog = SkyCoord(ra=ra2*u.degree, dec=dec2*u.degree, distance=distance2*u.kpc)
>>> idx, d2d, d3d = c.match_to_catalog_3d(catalog)
```

Now `idx` are indices into `catalog` that are the closest objects to each of the coordinates in `c`, `d2d` are the on-sky distances between them, and `d3d` are the 3-dimensional distances. Because coordinate objects support indexing, `idx` enables easy access to the matched set of coordinates in the catalog:

```
>>> matches = catalog[idx]
>>> (matches.separation_3d(c) == d3d).all()
True
>>> dra = (matches.ra - c.ra).arcmin
>>> ddec = (matches.dec - c.dec).arcmin
```

This functionality can also be accessed from the `match_coordinates_sky()` and `match_coordinates_3d()` functions. These will work on either `SkyCoord` objects *or* the lower-level frame classes:

```
>>> from astropy.coordinates import match_coordinates_sky
>>> idx, d2d, d3d = match_coordinates_sky(c, catalog)
>>> idx, d2d, d3d = match_coordinates_sky(c.frame, catalog.frame)
```

10.4.6 Using and Designing Coordinate Representations

As described in the *Overview of astropy.coordinates concepts*, the actual coordinate data in `astropy.coordinates` frames is represented via “Representation classes”. These can be used to store 3-d coordinates in various representations, such as cartesian, spherical polar, cylindrical, and so on. The built-in representation classes are:

- `CartesianRepresentation`: cartesian coordinates `x`, `y`, and `z`
- `SphericalRepresentation`: spherical polar coordinates represented by a longitude (`lon`), a latitude (`lat`), and a distance (`distance`). The latitude is a value ranging from -90 to 90 degrees.
- `UnitSphericalRepresentation`: spherical polar coordinates on a unit sphere, represented by a longitude (`lon`) and latitude (`lat`)
- `PhysicsSphericalRepresentation`: spherical polar coordinates, represented by an inclination (`theta`) and azimuthal angle (`phi`), and radius `r`. The inclination goes from 0 to 180 degrees, and is related to the latitude in the `SphericalRepresentation` by `theta = 90 deg - lat`.
- `CylindricalRepresentation`: cylindrical polar coordinates, represented by a cylindrical radius (`rho`), azimuthal angle (`phi`), and height (`z`).

Instantiating and converting

Representation classes should be instantiated with `Quantity` objects:

```
>>> from astropy import units as u
>>> from astropy.coordinates.representation import CartesianRepresentation
>>> car = CartesianRepresentation(3 * u.kpc, 5 * u.kpc, 4 * u.kpc)
>>> car
<CartesianRepresentation x=3.0 kpc, y=5.0 kpc, z=4.0 kpc>
```

Representations can be converted to other representations using the `represent_as` method:

```
>>> from astropy.coordinates.representation import SphericalRepresentation, CylindricalRepresentation
>>> sph = car.represent_as(SphericalRepresentation)
>>> sph
<SphericalRepresentation lon=1.03037682652 rad, lat=0.601264216679 rad, distance=7.07106781187 kpc>
>>> cyl = car.represent_as(CylindricalRepresentation)
>>> cyl
<CylindricalRepresentation rho=5.83095189485 kpc, phi=1.03037682652 rad, z=4.0 kpc>
```

All representations can be converted to each other without loss of information, with the exception of `UnitSphericalRepresentation`. This class is used to store the longitude and latitude of points but does not contain any distance to the points, and assumes that they are located on a unit and dimensionless sphere:

```
>>> from astropy.coordinates.representation import UnitSphericalRepresentation
>>> sph_unit = car.represent_as(UnitSphericalRepresentation)
>>> sph_unit
<UnitSphericalRepresentation lon=1.03037682652 rad, lat=0.601264216679 rad>
```

Converting back to cartesian, the absolute scaling information has been removed, and the points are still located on a unit sphere:

```
>>> sph_unit = car.represent_as(UnitSphericalRepresentation)
>>> sph_unit.represent_as(CartesianRepresentation)
<CartesianRepresentation x=0.4242... , y=0.7071... , z=0.5656... >
```

Array values

Array `Quantity` objects can also be passed to representations:

```
>>> import numpy as np
>>> x = np.random.random(100)
>>> y = np.random.random(100)
>>> z = np.random.random(100)
>>> car_array = CartesianRepresentation(x * u.m, y * u.m, z * u.m)
>>> car_array
<CartesianRepresentation (x, y, z) in m
  [(0.7093..., 0.7788..., 0.3842...),
   (0.8434..., 0.4543..., 0.9579...),
   ...
   (0.0179..., 0.8587..., 0.4916...),
   (0.0207..., 0.3355..., 0.2799...)]>
```

Creating your own representations

To create your own representation class, your class must inherit from the `BaseRepresentation` class. In addition the following must be defined:

- `__init__` method:
Has a signature like `__init__(self, comp1, comp2, comp3, copy=True)` for inputting the representation component values.
- `from_cartesian` class method:
Takes a `CartesianRepresentation` object and returns an instance of your class.
- `to_cartesian` method:
Returns a `CartesianRepresentation` object.
- `components` property:
Returns a tuple of the names of the coordinate components (such as `x`, `lon`, and so on).
- `attr_classes` class attribute (`OrderedDict`):
Defines the initializer class for each component. In most cases this class should be derived from `Quantity`. In particular these class initializers must take the value as the first argument and accept a `unit` keyword which takes a `Unit` initializer or `None` to indicate no unit. Also note that the keys of this dictionary are treated as the names of the components for this representation, with the default ordered given in the order they appear as keys.
- `recommended_units` dictionary (optional):
Maps component names to the recommended unit to convert the values of that component to. Can be `None` (or missing) to indicate there is no preferred unit. If this dictionary is not defined, no conversion of components to particular units will occur.

In pseudo-code, this means that your class will look like:

```
class MyRepresentation(BaseRepresentation):

    attr_classes = OrderedDict([('comp1', ComponentClass1),
                              ('comp2', ComponentClass2),
                              ('comp3', ComponentClass3)])

    # recommended_units is optional
    recommended_units = {'comp1': u.unit1, 'comp2': u.unit2, 'comp3': u.unit3}

    def __init__(self, ...):
        ...

    @classmethod
    def from_cartesian(self, cartesian):
        ...
        return MyRepresentation(...)

    def to_cartesian(self):
        ...
        return CartesianRepresentation(...)

    @property
    def components(self):
        return 'comp1', 'comp2', 'comp3'
```

Once you do this, you will then automatically be able to call `represent_as` to convert other representations to/from your representation class. Your representation will also be available for use in `SkyCoord` and all frame classes.

10.4.7 Using and Designing Coordinate Frames

In `astropy.coordinates`, as outlined in the *Overview of astropy.coordinates concepts*, subclasses of `BaseCoordinateFrame` (“frame classes”) define particular coordinate frames. They can (but do not *have* to) contain representation objects storing the actual coordinate data. The actual coordinate transformations are defined as functions that transform representations between frame classes. This approach serves to separate high-level user functionality (see *Using the SkyCoord High-level Class*) and details of how the coordinates are actually stored (see *Using and Designing Coordinate Representations*) from the definition of frames and how they are transformed.

Using Frame Objects

Frames without Data

Frame objects have two distinct (but related) uses. The first is storing the information needed to uniquely define a frame (e.g., equinox, observation time). This information is stored on the frame objects as (read-only) Python attributes, which are set with the object is first created:

```
>>> from astropy.coordinates import ICRS, FK5
>>> FK5(equinox='J1975')
<FK5 Frame: equinox=J1975.000>
>>> ICRS() # has no attributes
<ICRS Frame>
>>> FK5() # uses default equinox
<FK5 Frame: equinox=J2000.000>
```

The specific names of attributes available for a particular frame (and their default values) are available as the class method `get_frame_attr_names`:

```
>>> FK5.get_frame_attr_names()
OrderedDict([('equinox', <Time object: scale='utc' format='jyear_str' value=J2000.000>)])
```

You can access any of the attributes on a frame by using standard Python attribute access. Note that for cases like `equinox`, which are time inputs, if you pass in any unambiguous time string, it will be converted into a `Time` object with UTC scale (see *Inferring input format*):

```
>>> f = FK5(equinox='J1975')
>>> f.equinox
<Time object: scale='utc' format='jyear_str' value=J1975.000>
>>> f = FK5(equinox='2011-05-15T12:13:14')
>>> f.equinox
<Time object: scale='utc' format='isot' value=2011-05-15T12:13:14.000>
```

Frames with Data

The second use for frame objects is to store actual realized coordinate data for frames like those described above. In this use, it is similar to the `SkyCoord` class, and in fact, the `SkyCoord` class internally uses the frame classes as its implementation. However, the frame classes have fewer “convenience” features, thereby keeping the implementation of frame classes simple. As such, they are created similarly to `SkyCoord` object. The simplest way is to use with keywords appropriate for the frame (e.g. `ra` and `dec` for equatorial systems):

```
>>> from astropy import units as u
>>> ICRS(ra=1.1*u.deg, dec=2.2*u.deg)
<ICRS Coordinate: ra=1.1 deg, dec=2.2 deg>
>>> FK5(ra=1.1*u.deg, dec=2.2*u.deg, equinox='J1975')
<FK5 Coordinate: equinox=J1975.000, ra=1.1 deg, dec=2.2 deg>
```

These same attributes can be used to access the data in the frames, as `Angle` objects (or `Angle` subclasses):

```
>>> coo = ICRS(ra=1.1*u.deg, dec=2.2*u.deg)
>>> coo.ra
<Longitude 1.1 deg>
>>> coo.ra.value
1.1
>>> coo.ra.to(u.hourangle)
<Longitude 0.07333333333333335 hourangle>
```

You can use the `representation` attribute in conjunction with the `representation_component_names` attribute to figure out what keywords are accepted by a particular class object. The former will be the representation class the system is expressed in (e.g., spherical for equatorial frames), and the latter will be a dictionary mapping names for that frame to the attribute name on the representation class:

```
>>> import astropy.units as u
>>> icrs = ICRS(1*u.deg, 2*u.deg)
>>> icrs.representation
<class 'astropy.coordinates.representation.SphericalRepresentation'>
>>> icrs.representation_component_names
OrderedDict([(u'ra', u'lon'), (u'dec', u'lat'), (u'distance', u'distance')])
```

The representation of the coordinate object can be changed, as shown below. This actually does *nothing* to the object internal data which stores the coordinate values, but it changes the external view of that data in two ways: (1) the object prints itself in accord with the new representation, and (2) the available attributes change to match those of the new representation (e.g. from `ra`, `dec`, `distance` to `x`, `y`, `z`). Setting the `representation` thus changes a *property* of the object (how it appears) without changing the intrinsic object itself which represents a point in 3d space.

```
>>> from astropy.coordinates import CartesianRepresentation
>>> icrs.representation = CartesianRepresentation
>>> icrs
<ICRS Coordinate: x=0.999238614955 , y=0.0174417749028 , z=0.0348994967025 >
>>> icrs.x
<Quantity 0.9992386149554826>
```

The representation can also be set at the time of creating a coordinate and affects the set of keywords used to supply the coordinate data. For example to create a coordinate with cartesian data do:

```
>>> ICRS(x=1*u.kpc, y=2*u.kpc, z=3*u.kpc, representation=CartesianRepresentation)
<ICRS Coordinate: x=1.0 kpc, y=2.0 kpc, z=3.0 kpc>
```

For more information about the use of representations in coordinates see the [Representations](#) section, and for details about the representations themselves see [Using and Designing Coordinate Representations](#).

There are two other ways to create frame classes with coordinates. A representation class can be passed in directly at creation, along with any frame attributes required:

```
>>> from astropy.coordinates import SphericalRepresentation
>>> rep = SphericalRepresentation(lon=1.1*u.deg, lat=2.2*u.deg, distance=3.3*u.kpc)
>>> FK5(rep, equinox='J1975')
<FK5 Coordinate: equinox=J1975.000, ra=1.1 deg, dec=2.2 deg, distance=3.3 kpc>
```

A final way is to create a frame object from an already existing frame (either one with or without data), using the `realize_frame` method. This will yield a frame with the same attributes, but new data:

```
>>> f1 = FK5(equinox='J1975')
>>> f1
<FK5 Frame: equinox=J1975.000>
>>> rep = SphericalRepresentation(lon=1.1*u.deg, lat=2.2*u.deg, distance=3.3*u.kpc)
```

```
>>> f1.realize_frame(rep)
<FK5 Coordinate: equinox=J1975.000, ra=1.1 deg, dec=2.2 deg, distance=3.3 kpc>
```

You can check if a frame object has data using the `has_data` attribute, and if it is preset, it can be accessed from the `data` attribute:

```
>>> ICRS().has_data
False
>>> cooi = ICRS(ra=1.1*u.deg, dec=2.2*u.deg)
>>> cooi.has_data
True
>>> cooi.data
<UnitSphericalRepresentation lon=1.1 deg, lat=2.2 deg>
```

All of the above methods can also accept array data (or other Python sequences) to create arrays of coordinates:

```
>>> ICRS(ra=[1.5, 2.5]*u.deg, dec=[3.5, 4.5]*u.deg)
<ICRS Coordinate: (ra, dec) in deg
  [(1.5, 3.5), (2.5, 4.5)]>
```

If you pass in mixed arrays and scalars, the arrays will be broadcast over the scalars appropriately:

```
>>> ICRS(ra=[1.5, 2.5]*u.deg, dec=[3.5, 4.5]*u.deg, distance=5*u.kpc)
<ICRS Coordinate: (ra, dec, distance) in (deg, deg, kpc)
  [(1.5, 3.5, 5.0), (2.5, 4.5, 5.0)]>
```

An additional operation that may be useful is the ability to extract the data in different representations. E.g., to get the Cartesian form of an ICRS coordinate:

```
>>> from astropy.coordinates import CartesianRepresentation
>>> cooi = ICRS(ra=0*u.deg, dec=45*u.deg, distance=10*u.pc)
>>> cooi.represent_as(CartesianRepresentation)
<CartesianRepresentation x=7.07106781187 pc, y=0.0 pc, z=7.07106781187 pc>
```

Transforming between Frames

To transform a frame object with data into another frame, use the `transform_to` method of an object, and provide it the frame you wish to transform to. This frame can either be a frame *class*, in which case the default attributes will be used, or a frame object (with or without data):

```
>>> cooi = ICRS(1.5*u.deg, 2.5*u.deg)
>>> cooi.transform_to(FK5)
<FK5 Coordinate: equinox=J2000.000, ra=1.50000660527 deg, dec=2.50000238221 deg>
>>> cooi.transform_to(FK5(equinox='J1975'))
<FK5 Coordinate: equinox=J1975.000, ra=1.17960348105 deg, dec=2.36085320826 deg>
```

The *Reference/API* includes a list of all of the frames built into `astropy.coordinates`, as well as the defined transformations between them. Any transformation that has a valid path, even if it passes through other frames, can be transformed to. To programmatically check for or manipulate transformations, see the `TransformGraph` documentation.

Defining a New Frame

Users can add new coordinate frames by creating new classes that are subclasses of `BaseCoordinateFrame`. Detailed instructions for subclassing are in the docstrings for that class. The key aspects are to define the class attributes `default_representation` and `frame_specific_representation_info` along with frame

attributes as `FrameAttribute` class instances (or subclasses like `TimeFrameAttribute`). If these are defined, there is often no need to define an `__init__` function, as the initializer in `BaseCoordinateFrame` will probably behave the way you want. As an example:

```
>>> from astropy.coordinates import BaseCoordinateFrame, FrameAttribute, TimeFrameAttribute, Representation
>>> class MyFrame(BaseCoordinateFrame):
...     # Specify how coordinate values are represented when outputted
...     default_representation = SphericalRepresentation
...
...     # Specify overrides to the default names and units for all available
...     # representations (subclasses of BaseRepresentation).
...     frame_specific_representation_info = {
...         'spherical': [RepresentationMapping(reprname='lon', framename='R', defaultunit=u.rad),
...                       RepresentationMapping(reprname='lat', framename='D', defaultunit=u.rad),
...                       RepresentationMapping(reprname='distance', framename='DIST', defaultunit=u.rad)],
...         'unitspherical': [RepresentationMapping(reprname='lon', framename='R', defaultunit=u.rad),
...                            RepresentationMapping(reprname='lat', framename='D', defaultunit=u.rad)],
...         'cartesian': [RepresentationMapping(reprname='x', framename='X'),
...                       RepresentationMapping(reprname='y', framename='Y'),
...                       RepresentationMapping(reprname='z', framename='Z')]
...     }
...
...     # Specify frame attributes required to fully specify the frame
...     location = FrameAttribute(default=None)
...     equinox = TimeFrameAttribute(default='B1950')
...     obstime = TimeFrameAttribute(default=None, secondary_attribute='equinox')

>>> c = MyFrame(R=10*u.deg, D=20*u.deg)
>>> c
<MyFrame Coordinate: location=None, equinox=B1950.000, obstime=B1950.000, R=0.174532925199 rad, D=0.349065850424 rad>
>>> c.equinox
<Time object: scale='utc' format='byear_str' value=B1950.000>
```

You can also define arbitrary methods for any added functionality you want your frame to have that's unique to that frame. These methods will be available in any `SkyCoord` that is created using your user-defined frame.

For examples of defining frame classes, the first place to look is probably the source code for the frames that are included in `astropy` (available at `astropy.coordinates.builtin_frames`). These are not “magic” in any way, and use all the same API and features available to user-created frames. A more annotated example is also available in the *Example: Defining A Coordinate Frame for the Sgr Dwarf* documentation section.

Defining Transformations

A frame may not be too useful without a way to transform coordinates defined in it to or from other frames. Fortunately, `astropy.coordinates` provides a framework to do just that. The key concept for these transformations is the frame transform graph, available as `astropy.coordinates.frame_transform_graph`, an instance of the `TransformGraph` class. This graph (in the “graph theory” sense, not “plot”), stores all the transformations between all of the builtin frames, as well as tools for finding shortest paths through this graph to transform from any frame to any other. All of the power of this graph is available to user-created frames, meaning that once you define even one transform from your frame to some frame in the graph, coordinates defined in your frame can be transformed to *any* other frame in the graph.

The transforms themselves are represented as `CoordinateTransform` objects or their subclasses. The useful subclasses/types of transformations are:

- `FunctionTransform`

A transform that is defined as a function that takes a frame object of one frame class and returns an object of another class.

- `StaticMatrixTransform`
- `DynamicMatrixTransform`

These are both for transformations defined as a 3x3 matrix transforming the Cartesian representation of one frame into the target frame’s Cartesian representation. The static version is for the case where the matrix is independent of the frame attributes (e.g., the ICRS->FK5 transformation, because ICRS has no frame attributes). The dynamic case is for transformations where the transformation matrix depends on the frame attributes of either the to or from frame.

Generally, it is not necessary to use these classes directly. Instead, use methods on `frame_transform_graph` that can be used as function decorators. Then just define functions that either do the actual transformation (for `FunctionTransform`), or that compute the necessary transformation matrices to transform. Then decorate the functions to register these transformations with the frame transform graph:

```
from astropy.coordinates import frame_transform_graph

@frame_transform_graph.transform(DynamicMatrixTransform, ICRS, FK5)
def icrs_to_fk5(icrscoord, fk5frame):
    ...

@frame_transform_graph.transform(DynamicMatrixTransform, FK5, ICRS)
def fk5_to_icrs(fk5coord, icrsframe):
    ...
```

If the transformation to your coordinate frame of interest is not representable by a matrix operation, you can also specify a function to do the actual transformation, and pass the `FunctionTransform` class to the transform graph decorator instead:

```
@frame_transform_graph.transform(FunctionTransform, FK4NoETerms, FK4)
def fk4_no_e_to_fk4(fk4noecoord, fk4frame):
    ...
```

Furthermore, the `frame_transform_graph` does some caching and optimization to speed up transformations after the first attempt to go from one frame to another, and shortcuts steps where relevant (for example, combining multiple static matrix transforms into a single matrix). Hence, in general, it is better to define whatever are the most natural transformations for a user-defined frame, rather than worrying about optimizing or caching a transformation to speed up the process.

10.4.8 Example: Defining A Coordinate Frame for the Sgr Dwarf

This document describes in detail how to subclass and define a custom spherical coordinate frame, as discussed in *Using and Designing Coordinate Frames* and the docstring for `BaseCoordinateFrame`. In this example, we will define a coordinate system defined by the plane of orbit of the Sagittarius Dwarf Galaxy (hereafter Sgr; as defined in Majewski et al. 2003). The Sgr coordinate system is often referred to in terms of two angular coordinates, Λ , B .

We need to define a subclass of `BaseCoordinateFrame` that knows the names and units of the coordinate system angles in each of the supported representations. In this case we support `SphericalRepresentation` with “Lambda” and “Beta”. Then we have to define the transformation from this coordinate system to some other built-in system. Here we will use Galactic coordinates, represented by the `Galactic` class.

The first step is to create a new class, which we’ll call `Sagittarius` and make it a subclass of `BaseCoordinateFrame`:

```
import numpy as np
from numpy import cos, sin

from astropy.coordinates import frame_transform_graph
from astropy.coordinates.angles import rotation_matrix
import astropy.coordinates as coord
import astropy.units as u

class Sagittarius(coord.BaseCoordinateFrame):
    """
    A Heliocentric spherical coordinate system defined by the orbit
    of the Sagittarius dwarf galaxy, as described in
    http://adsabs.harvard.edu/abs/2003ApJ...599.1082M
    and further explained in
    http://www.astro.virginia.edu/~srm4n/Sgr/.

    Parameters
    -----
    representation : `BaseRepresentation` or None
        A representation object or None to have no data (or use the other keywords)
    Lambda : `Angle`, optional, must be keyword
        The longitude-like angle corresponding to Sagittarius' orbit.
    Beta : `Angle`, optional, must be keyword
        The latitude-like angle corresponding to Sagittarius' orbit.
    distance : `Quantity`, optional, must be keyword
        The Distance for this object along the line-of-sight.

    """
    default_representation = coord.SphericalRepresentation

    frame_specific_representation_info = {
        'spherical': [coord.RepresentationMapping('lon', 'Lambda'),
                     coord.RepresentationMapping('lat', 'Beta'),
                     coord.RepresentationMapping('distance', 'distance')],
        'unitspherical': [coord.RepresentationMapping('lon', 'Lambda'),
                          coord.RepresentationMapping('lat', 'Beta')]
    }
```

Line by line, the first few are simply imports. Next we define the class as a subclass of `BaseCoordinateFrame`. Then we include a descriptive docstring. The final lines are class-level attributes that specify the default representation for the data and mappings from the attribute names used by representation objects to the names that are to be used by `Sagittarius`. In this case we override the names in the spherical representations but don't do anything with other representations like cartesian or cylindrical.

Next we have to define the transformation to some other built-in coordinate system; we will use Galactic coordinates. We can do this by defining functions that return transformation matrices, or by simply defining a function that accepts a coordinate and returns a new coordinate in the new system. We'll start by constructing the rotation matrix, using the helper function `rotation_matrix`:

```
# Define the Euler angles (from Law & Majewski 2010)
SGR_PHI = np.radians(180+3.75)
SGR_THETA = np.radians(90-13.46)
SGR_PSI = np.radians(180+14.111534)

# Generate the rotation matrix using the x-convention (see Goldstein)
D = rotation_matrix(SGR_PHI, "z", unit=u.radian)
C = rotation_matrix(SGR_THETA, "x", unit=u.radian)
B = rotation_matrix(SGR_PSI, "z", unit=u.radian)
```

```
SGR_MATRIX = np.array(B.dot(C).dot(D))
```

This is done at the module level, since it will be used by both the transformation from Sgr to Galactic as well as the inverse from Galactic to Sgr. Now we can define our first transformation function:

```
# Galactic to Sgr coordinates
@frame_transform_graph.transform(coord.FunctionTransform, coord.Galactic, Sagittarius)
def galactic_to_sgr(gal_coord, sgr_frame):
    """ Compute the transformation from Galactic spherical to
        heliocentric Sgr coordinates.
    """

    l = np.atleast_1d(gal_coord.l.radian)
    b = np.atleast_1d(gal_coord.b.radian)

    X = np.cos(b)*np.cos(l)
    Y = np.cos(b)*np.sin(l)
    Z = np.sin(b)

    # Calculate X,Y,Z,distance in the Sgr system
    Xs, Ys, Zs = SGR_MATRIX.dot(np.array([X, Y, Z]))
    Zs = -Zs

    # Calculate the angular coordinates lambda,beta
    Lambda = np.arctan2(Ys,Xs)*u.radian
    Lambda[Lambda < 0] = Lambda[Lambda < 0] + 2.*np.pi*u.radian
    Beta = np.arcsin(Zs/np.sqrt(Xs*Xs+Ys*Ys+Zs*Zs))*u.radian

    return Sagittarius(Lambda=Lambda, Beta=Beta,
                       distance=gal_coord.distance)
```

The decorator `@frame_transform_graph.transform(coord.FunctionTransform, coord.Galactic, Sagittarius)` registers this function on the `frame_transform_graph` as a transformation. Inside the function, we simply follow the same procedure as detailed by David Law's [transformation code](#). Note that in this case, both coordinate systems are heliocentric, so we can simply copy any distance from the Galactic object.

We then register the inverse transformation by using the transpose of the rotation matrix (which is faster to compute than the inverse):

```
# Sgr to Galactic coordinates
@frame_transform_graph.transform(coord.FunctionTransform, Sagittarius, coord.Galactic)
def sgr_to_galactic(sgr_coord, gal_frame):
    """ Compute the transformation from heliocentric Sgr coordinates to
        spherical Galactic.
    """

    L = np.atleast_1d(sgr_coord.Lambda.radian)
    B = np.atleast_1d(sgr_coord.Beta.radian)

    Xs = cos(B)*cos(L)
    Ys = cos(B)*sin(L)
    Zs = sin(B)
    Zs = -Zs

    X, Y, Z = SGR_MATRIX.T.dot(np.array([Xs, Ys, Zs]))

    l = np.arctan2(Y,X)*u.radian
    b = np.arcsin(Z/np.sqrt(X*X+Y*Y+Z*Z))*u.radian
```

```
l[l<=0] += 2*np.pi*u.radian

    return coord.Galactic(l=l, b=b, distance=sgr_coord.distance)
```

Now that we've registered these transformations between Sagittarius and `Galactic`, we can transform between *any* coordinate system and Sagittarius (as long as the other system has a path to transform to `Galactic`). For example, to transform from ICRS coordinates to Sagittarius, we simply:

```
>>> import astropy.units as u
>>> import astropy.coordinates as coord
>>> icrs = coord.ICRS(280.161732*u.degree, 11.91934*u.degree)
>>> icrs.transform_to(Sagittarius)
<Sagittarius Coordinate: (Lambda, Beta, distance) in (deg, deg, )
    (346.818273..., -39.283667..., 1.0)>
```

The complete code for the above example is included below for reference.

See Also

- Majewski et al. 2003, “A Two Micron All Sky Survey View of the Sagittarius Dwarf Galaxy. I. Morphology of the Sagittarius Core and Tidal Arms”, <http://arxiv.org/abs/astro-ph/0304198>
- Law & Majewski 2010, “The Sagittarius Dwarf Galaxy: A Model for Evolution in a Triaxial Milky Way Halo”, <http://arxiv.org/abs/1003.1132>
- David Law’s Sgr info page <http://www.astro.virginia.edu/~srm4n/Sgr/>

Complete Code for Example

```
1 # coding: utf-8
2
3 """ Astropy coordinate class for the Sagittarius coordinate system """
4
5 from __future__ import division, print_function
6
7 __author__ = "adrn <adrn@astro.columbia.edu>"
8
9 # Third-party
10 import numpy as np
11 from numpy import cos, sin
12
13 from astropy.coordinates import frame_transform_graph
14 from astropy.coordinates.angles import rotation_matrix
15 import astropy.coordinates as coord
16 import astropy.units as u
17
18
19 __all__ = ["Sagittarius"]
20
21
22 class Sagittarius(coord.BaseCoordinateFrame):
23     """
24     A Heliocentric spherical coordinate system defined by the orbit
25     of the Sagittarius dwarf galaxy, as described in
26     http://adsabs.harvard.edu/abs/2003ApJ...599.1082M
27     and further explained in
28     http://www.astro.virginia.edu/~srm4n/Sgr/.
```

```

29
30 Parameters
31 -----
32 representation : `BaseRepresentation` or None
33     A representation object or None to have no data (or use the other keywords)
34 Lambda : `Angle`, optional, must be keyword
35     The longitude-like angle corresponding to Sagittarius' orbit.
36 Beta : `Angle`, optional, must be keyword
37     The latitude-like angle corresponding to Sagittarius' orbit.
38 distance : `Quantity`, optional, must be keyword
39     The Distance for this object along the line-of-sight.
40
41 """
42 default_representation = coord.SphericalRepresentation
43
44 frame_specific_representation_info = {
45     'spherical': [coord.RepresentationMapping('lon', 'Lambda'),
46                 coord.RepresentationMapping('lat', 'Beta'),
47                 coord.RepresentationMapping('distance', 'distance')],
48     'unitspherical': [coord.RepresentationMapping('lon', 'Lambda'),
49                     coord.RepresentationMapping('lat', 'Beta')]
50 }
51
52 # Define the Euler angles (from Law & Majewski 2010)
53 phi = np.radians(180+3.75)
54 theta = np.radians(90-13.46)
55 psi = np.radians(180+14.111534)
56
57 # Generate the rotation matrix using the x-convention (see Goldstein)
58 D = rotation_matrix(phi, "z", unit=u.radian)
59 C = rotation_matrix(theta, "x", unit=u.radian)
60 B = rotation_matrix(psi, "z", unit=u.radian)
61 sgr_matrix = np.array(B.dot(C).dot(D))
62
63
64 # Galactic to Sgr coordinates
65 @frame_transform_graph.transform(coord.FunctionTransform, coord.Galactic, Sagittarius)
66 def galactic_to_sgr(gal_coord, sgr_frame):
67     """ Compute the transformation from Galactic spherical to
68         heliocentric Sgr coordinates.
69     """
70
71     l = np.atleast_1d(gal_coord.l.radian)
72     b = np.atleast_1d(gal_coord.b.radian)
73
74     X = cos(b)*cos(l)
75     Y = cos(b)*sin(l)
76     Z = sin(b)
77
78     # Calculate X,Y,Z,distance in the Sgr system
79     Xs, Ys, Zs = sgr_matrix.dot(np.array([X, Y, Z]))
80     Zs = -Zs
81
82     # Calculate the angular coordinates lambda,beta
83     Lambda = np.arctan2(Ys, Xs)*u.radian
84     Lambda[Lambda < 0] = Lambda[Lambda < 0] + 2.*np.pi*u.radian
85     Beta = np.arcsin(Zs/np.sqrt(Xs*Xs+Ys*Ys+Zs*Zs))*u.radian
86

```

```
87     return Sagittarius(Lambda=Lambda, Beta=Beta,
88                        distance=gal_coord.distance)
89
90
91 # Sgr to Galactic coordinates
92 @frame_transform_graph.transform(coord.FunctionTransform, Sagittarius, coord.Galactic)
93 def sgr_to_galactic(sgr_coord, gal_frame):
94     """ Compute the transformation from heliocentric Sgr coordinates to
95         spherical Galactic.
96     """
97     L = np.atleast_1d(sgr_coord.Lambda.radian)
98     B = np.atleast_1d(sgr_coord.Beta.radian)
99
100     Xs = cos(B)*cos(L)
101     Ys = cos(B)*sin(L)
102     Zs = sin(B)
103     Zs = -Zs
104
105     X, Y, Z = sgr_matrix.T.dot(np.array([Xs, Ys, Zs]))
106
107     l = np.arctan2(Y, X)*u.radian
108     b = np.arcsin(Z/np.sqrt(X*X+Y*Y+Z*Z))*u.radian
109
110     l[l<0] += 2*np.pi*u.radian
111
112     return coord.Galactic(l=l, b=b, distance=sgr_coord.distance)
113
114 if __name__ == "__main__":
115     # Example use case for our newly defined coordinate class
116     icrs = coord.ICRS(152.88572*u.degree, 11.57281*u.degree)
117     sgr = icrs.transform_to(Sagittarius)
118     print(sgr)
```

10.4.9 Important Definitions

For reference, below, we define some key terms as they are used in `coordinates`, due to some ambiguities that exist in the colloquial use of these terms. Chief among these terms is the concept of a “coordinate system.” To some members of the community, “coordinate system” means the *representation* of a point in space, e.g., “Cartesian coordinate system” is different from “Spherical polar coordinate system”. Another use of “coordinate system” is to mean a unique reference frame with a particular set of reference points, e.g., “the ICRS coordinate system” or the “J2000 coordinate system.” This second meaning is further complicated by the fact that such systems use quite different ways of defining a frame.

Because of the likelihood of confusion between these meanings of “coordinate system”, `coordinates` avoids this term wherever possible, and instead adopts the following terms (loosely inspired by the IAU2000 resolutions on celestial coordinate systems):

- A “Coordinate Representation” is a particular way of describing a unique point in a vector space. (Here, this means three-dimensional space, but future extensions might have different dimensionality, particularly if relativistic effects are desired.) Examples include Cartesian coordinates, cylindrical polar, or latitude/longitude spherical polar coordinates.
- A “Reference System” is a scheme for orienting points in a space and describing how they transforms to other systems. Examples include the ICRS, equatorial coordinates with mean equinox, or the WGS84 geoid for latitude/longitude on the Earth.
- A “Coordinate Frame”, “Reference Frame”, or just “Frame” is a specific realization of a reference system - e.g.,

the ICRF, or J2000 equatorial coordinates. For some systems, there may be only one meaningful frame, while others may have many different frames (differentiated by something like a different equinox, or a different set of reference points).

- A “Coordinate” is a combination of all of the above that specifies a unique point.

In addition, another resource for the capabilities of this package is the `astropy.coordinates.tests.test_api_ape5` testing file. It showcases most of the major capabilities of the package, and hence is a useful supplement to this document. You can see it by either looking at it directly if you downloaded a copy of the astropy source code, or typing the following in an IPython session:

```
In [1]: from astropy.coordinates.tests import test_api_ape5
In [2]: test_api_ape5??
```

10.5 Migrating from pre-v0.4 coordinates

For typical users, the major change is that the recommended way to use coordinate functionality is via the `SkyCoord` class, instead of classes like `ICRS` classes (now called “frame classes”).

For most users of pre-v0.4 coordinates, this means that the best way to adapt old code to the new framework is to change code like:

```
>>> from astropy import units as u
>>> from astropy.coordinates import ICRS # or FK5, or Galactic, or similar
>>> coordinate = ICRS(123.4*u.deg, 56.7*u.deg)
```

to instead be:

```
>>> from astropy import units as u
>>> from astropy.coordinates import SkyCoord
>>> coordinate = SkyCoord(123.4*u.deg, 56.7*u.deg, frame='icrs')
```

Note that usage like:

```
>>> coordinate = ICRS(123.4, 56.7, unit=('deg', 'deg')) # NOT RECOMMENDED!
```

will continue to work in v0.4, but will yield a `SkyCoord` instead of an `ICRS` object (the former behaves more like the pre-v0.4 `ICRS`). This compatibility feature will issue a deprecation warning, and will be removed in the next major version, so you should update your code to use `SkyCoord` directly by the next release.

Users should also be aware that if they continue to use the first form (directly creating `ICRS` frame objects), old code may still work if it uses basic coordinate functionality, but many of the convenience functions like catalog matching or attribute-based transforms like `coordinate.galactic` will no longer work. These features are now all in `SkyCoord`.

For advanced users or developers who have defined their own coordinates, take note that the extensive internal changes will require re-writing user-defined coordinate frames. The *Example: Defining A Coordinate Frame for the Sgr Dwarf* document has been updated for the new framework to provide a worked example of how custom coordinates work.

More detailed information about the new framework and using it to define custom coordinates is available at *Overview of astropy.coordinates concepts, Important Definitions, Defining a New Frame, and Creating your own representations*.

10.6 See Also

Some references particularly useful in understanding subtleties of the coordinate systems implemented here include:

- **Standards Of Fundamental Astronomy**
The definitive implementation of IAU-defined algorithms. The “SOFA Tools for Earth Attitude” document is particularly valuable for understanding the latest IAU standards in detail.
- **USNO Circular 179**
A useful guide to the IAU 2000/2003 work surrounding ICRS/IERS/CIRS and related problems in precision coordinate system work.
- **Meeus, J. “Astronomical Algorithms”**
A valuable text describing details of a wide range of coordinate-related problems and concepts.

10.7 Reference/API

10.7.1 astropy.coordinates Module

This subpackage contains classes and functions for celestial coordinates of astronomical objects. It also contains a framework for conversions between coordinate systems.

The diagram below shows all of the coordinate systems built into the `coordinates` package, their aliases (useful for converting other coordinates to them using attribute-style access) and the pre-defined transformations between them. The user is free to override any of these transformations by defining new transformations between these systems, but the pre-defined transformations should be sufficient for typical usage.

The graph also indicates the priority for each transformation as a number next to the arrow. These priorities are used to decide the preferred order when two transformation paths have the same number of steps. These priorities are defined such that the path with a *smaller* total priority is favored.

Functions

<code>cartesian_to_spherical(x, y, z)</code>	Converts 3D rectangular cartesian coordinates to spherical polar coordinates.
<code>get_icrs_coordinates(name)</code>	Retrieve an ICRS object by using an online name resolving service to resolve the name.
<code>match_coordinates_3d(matchcoord, catalogcoord)</code>	Finds the nearest 3-dimensional matches of a coordinate or coordinates in a set of coordinates.
<code>match_coordinates_sky(matchcoord, catalogcoord)</code>	Finds the nearest on-sky matches of a coordinate or coordinates in a set of coordinates.
<code>spherical_to_cartesian(r, lat, lon)</code>	Converts spherical polar coordinates to rectangular cartesian coordinates.

`cartesian_to_spherical`

`astropy.coordinates.cartesian_to_spherical(x, y, z)`

Converts 3D rectangular cartesian coordinates to spherical polar coordinates.

Note that the resulting angles are latitude/longitude or elevation/azimuthal form. I.e., the origin is along the equator rather than at the north pole.

Note: This is a low-level function used internally in `astropy.coordinates`. It is provided for users if they really want to use it, but it is recommended that you use the `astropy.coordinates` coordinate systems.

Parameters

`x` : scalar or array-like

The first cartesian coordinate.

`y` : scalar or array-like

The second cartesian coordinate.

z : scalar or array-like

The third cartesian coordinate.

Returns

r : float or array

The radial coordinate (in the same units as the inputs).

lat : float or array

The latitude in radians

lon : float or array

The longitude in radians

get_icrs_coordinates

`astropy.coordinates.get_icrs_coordinates` (*name*)

Retrieve an ICRS object by using an online name resolving service to retrieve coordinates for the specified name. By default, this will search all available databases until a match is found. If you would like to specify the database, use the science state `astropy.coordinates.name_resolve.sesame_database`. You can also specify a list of servers to use for querying Sesame using the science state `astropy.coordinates.name_resolve.sesame_url`. This will try each one in order until a valid response is returned. By default, this list includes the main Sesame host and a mirror at vizier. The configuration item `astropy.utils.data.Conf.remote_timeout` controls the number of seconds to wait for a response from the server before giving up.

Parameters

name : str

The name of the object to get coordinates for, e.g. 'M42'.

Returns

coord : `astropy.coordinates.ICRS` object

The object's coordinates in the ICRS frame.

match_coordinates_3d

`astropy.coordinates.match_coordinates_3d` (*matchcoord*, *catalogcoord*, *nthneighbor=1*, *storekdtree=u'kdtree_3d'*)

Finds the nearest 3-dimensional matches of a coordinate or coordinates in a set of catalog coordinates.

This finds the 3-dimensional closest neighbor, which is only different from the on-sky distance if `distance` is set in either `matchcoord` or `catalogcoord`.

Parameters

matchcoord : `BaseCoordinateFrame` or `SkyCoord`

The coordinate(s) to match to the catalog.

catalogcoord : `BaseCoordinateFrame` or `SkyCoord`

The base catalog in which to search for matches. Typically this will be a coordinate object that is an array (i.e., `catalogcoord.isscalar == False`)

nthneighbor : int, optional

Which closest neighbor to search for. Typically 1 is desired here, as that is correct for matching one set of coordinates to another. The next likely use case is 2, for matching a coordinate catalog against *itself* (1 is inappropriate because each point will find itself as the closest match).

storekdtree : bool or str, optional

If a string, will store the KD-Tree used for the computation in the `catalogcoord`, as an attribute in `catalogcoord` with the provided name. This dramatically speeds up subsequent calls with the same catalog. If False, the KD-Tree is discarded after use.

Returns

idx : integer array

Indices into `catalogcoord` to get the matched points for each `matchcoord`. Shape matches `matchcoord`.

sep2d : `Angle`

The on-sky separation between the closest match for each `matchcoord` and the `matchcoord`. Shape matches `matchcoord`.

dist3d : `Quantity`

The 3D distance between the closest match for each `matchcoord` and the `matchcoord`. Shape matches `matchcoord`.

Notes

This function requires `SciPy` to be installed or it will fail.

`match_coordinates_sky`

`astropy.coordinates.match_coordinates_sky(matchcoord, catalogcoord, nthneighbor=1, storekdtree=u'kdtree_sky')`

Finds the nearest on-sky matches of a coordinate or coordinates in a set of catalog coordinates.

This finds the on-sky closest neighbor, which is only different from the 3-dimensional match if distance is set in either `matchcoord` or `catalogcoord`.

Parameters

matchcoord : `BaseCoordinateFrame` or `SkyCoord`

The coordinate(s) to match to the catalog.

catalogcoord : `BaseCoordinateFrame` or `SkyCoord`

The base catalog in which to search for matches. Typically this will be a coordinate object that is an array (i.e., `catalogcoord.isscalar == False`)

nthneighbor : int, optional

Which closest neighbor to search for. Typically 1 is desired here, as that is correct for matching one set of coordinates to another. The next likely use case is 2, for matching a coordinate catalog against *itself* (1 is inappropriate because each point will find itself as the closest match).

storekdtree : bool or str, optional

If a string, will store the KD-Tree used for the computation in the `catalogcoord`, as an attribute in `catalogcoord` with the provided name. This dramatically speeds up subsequent calls with the same catalog. If False, the KD-Tree is discarded after use.

Returns

idx : integer array

Indices into `catalogcoord` to get the matched points for each `matchcoord`. Shape matches `matchcoord`.

sep2d : `Angle`

The on-sky separation between the closest match for each `matchcoord` and the `matchcoord`. Shape matches `matchcoord`.

dist3d : `Quantity`

The 3D distance between the closest match for each `matchcoord` and the `matchcoord`. Shape matches `matchcoord`. If either `matchcoord` or `catalogcoord` don't have a distance, this is the 3D distance on the unit sphere, rather than a true distance.

Notes

This function requires `SciPy` to be installed or it will fail.

spherical_to_cartesian

`astropy.coordinates.spherical_to_cartesian` (*r*, *lat*, *lon*)

Converts spherical polar coordinates to rectangular cartesian coordinates.

Note that the input angles should be in latitude/longitude or elevation/azimuthal form. I.e., the origin is along the equator rather than at the north pole.

Note: This is a low-level function used internally in `astropy.coordinates`. It is provided for users if they really want to use it, but it is recommended that you use the `astropy.coordinates` coordinate systems.

Parameters

r : scalar or array-like

The radial coordinate (in the same units as the inputs).

lat : scalar or array-like

The latitude in radians

lon : scalar or array-like

The longitude in radians

Returns

x : float or array

The first cartesian coordinate.

y : float or array

The second cartesian coordinate.

z : float or array

The third cartesian coordinate.

Classes

<code>AltAz(*args, **kwargs)</code>	A coordinate or frame in the Altitude-Azimuth system (i.e., Horizontal coordinates).
<code>Angle</code>	One or more angular value(s) with units equivalent to radians or degrees.
<code>BaseCoordinateFrame(*args, **kwargs)</code>	The base class for coordinate frames.
<code>BaseRepresentation</code>	Base Representation object, for representing a point in a 3D coordinate system.
<code>BoundsError</code>	Raised when an angle is outside of its user-specified bounds.
<code>CartesianPoints(*args, **kwargs)</code>	Deprecated since version v0.4.
<code>CartesianRepresentation(x[, y, z, copy])</code>	Representation of points in 3D cartesian coordinates.
<code>CompositeTransform(transforms, fromsys, tosys)</code>	A transformation constructed by combining together a series of single-axis transformations.
<code>ConvertError</code>	Raised if a coordinate system cannot be converted to another.
<code>CoordinateTransform(fromsys, tosys[, ...])</code>	An object that transforms a coordinate from one system to another.
<code>CylindricalRepresentation(rho, phi, z[, copy])</code>	Representation of points in 3D cylindrical coordinates.
<code>Distance</code>	A one-dimensional distance.
<code>DynamicMatrixTransform(matrix_func, fromsys, ...)</code>	A coordinate transformation specified as a function that yields a 3 x 3 cartesian transformation matrix.
<code>EarthLocation</code>	Location on Earth.
<code>FK4(*args, **kwargs)</code>	A coordinate or frame in the FK4 system.
<code>FK4NoETerms(*args, **kwargs)</code>	A coordinate or frame in the FK4 system, but with the E-terms of aberration removed.
<code>FK5(*args, **kwargs)</code>	A coordinate or frame in the FK5 system.
<code>FrameAttribute([default, secondary_attribute])</code>	A non-mutable data descriptor to hold a frame attribute.
<code>FunctionTransform(func, fromsys, tosys[, ...])</code>	A coordinate transformation defined by a function that accepts a coordinate and returns a coordinate.
<code>Galactic(*args, **kwargs)</code>	Galactic Coordinates.
<code>GenericFrame(frame_attrs)</code>	A frame object that can't store data but can hold any arbitrary frame attributes.
<code>ICRS(*args, **kwargs)</code>	A coordinate or frame in the ICRS system.
<code>IllegalHourError(hour)</code>	Raised when an hour value is not in the range [0,24).
<code>IllegalHourWarning(hour[, alternativeactionstr])</code>	Raised when an hour value is 24.
<code>IllegalMinuteError(minute)</code>	Raised when an minute value is not in the range [0,60).
<code>IllegalMinuteWarning(minute[, ...])</code>	Raised when a minute value is 60.
<code>IllegalSecondError(second)</code>	Raised when an second value (time) is not in the range [0,60).
<code>IllegalSecondWarning(second[, ...])</code>	Raised when a second value is 60.
<code>Latitude</code>	Latitude-like angle(s) which must be in the range -90 to +90 deg.
<code>Longitude</code>	Longitude-like angle(s) which are wrapped within a contiguous 360 deg range.
<code>PhysicsSphericalRepresentation(phi, theta, r)</code>	Representation of points in 3D spherical coordinates (using the physics convention).
<code>RangeError</code>	Raised when some part of an angle is out of its valid range.
<code>RepresentationMapping</code>	This <code>namedtuple</code> is used with the <code>frame_specific_representation_mappings</code> attribute of <code>BaseCoordinateFrame</code> .
<code>SkyCoord(*args, **kwargs)</code>	High-level object providing a flexible interface for celestial coordinate representations.
<code>SphericalRepresentation(lon, lat, distance)</code>	Representation of points in 3D spherical coordinates.
<code>StaticMatrixTransform(matrix, fromsys, tosys)</code>	A coordinate transformation defined as a 3 x 3 cartesian transformation matrix.
<code>TimeFrameAttribute([default, ...])</code>	Frame attribute descriptor for quantities that are Time objects.
<code>TransformGraph()</code>	A graph representing the paths between coordinate frames.
<code>UnitSphericalRepresentation(lon, lat[, copy])</code>	Representation of points on a unit sphere.

AltAz

```
class astropy.coordinates.AltAz(*args, **kwargs)
```

```
    Bases: astropy.coordinates.BaseCoordinateFrame
```

```
    A coordinate or frame in the Altitude-Azimuth system (i.e., Horizontal coordinates).
```

Warning: The AltAz class currently does not support any transformations. In a future version, it will support the standard IAU2000 AltAz<->ICRS transformations. It is provided right now as a placeholder for storing as-observed horizontal coordinates.

Parameters

representation : `BaseRepresentation` or `None`

A representation object or `None` to have no data (or use the other keywords)

az : `Angle`, optional, must be keyword

The Azimuth for this object (`alt` must also be given and `representation` must be `None`).

alt : `Angle`, optional, must be keyword

The Altitude for this object (`az` must also be given and `representation` must be `None`).

distance : `Quantity`, optional, must be keyword

The Distance for this object along the line-of-sight.

Attributes Summary

<code>default_representation</code>	
<code>equinox</code>	Represent and manipulate times and dates for astronomy.
<code>frame_specific_representation_info</code>	
<code>location</code>	
<code>name</code>	<code>str(object) -> string</code>
<code>obstime</code>	

Attributes Documentation

default_representation

equinox = <Time object: scale='tai' format='byear_str' value=B1950.000>

frame_specific_representation_info

location = `None`

name = 'altaz'

obstime = `None`

Angle

class `astropy.coordinates.Angle`
 Bases: `astropy.units.quantity.Quantity`

One or more angular value(s) with units equivalent to radians or degrees.

An angle can be specified either as an array, scalar, tuple (see below), string, `Quantity` or another `Angle`.

The input parser is flexible and supports a variety of formats:

```
Angle('10.2345d')
Angle(['10.2345d', '-20d'])
Angle('1:2:30.43 degrees')
Angle('1 2 0 hours')
Angle(np.arange(1, 8), unit=u.deg)
Angle(u'1°2'3"')
Angle('1d2m3.4s')
Angle('-1h2m3s')
Angle((-1, 2, 3), unit=u.deg) # (d, m, s)
Angle(10.2345 * u.deg)
Angle(Angle(10.2345 * u.deg))
```

Parameters

angle : array, scalar, `Quantity`, `Angle`

The angle value. If a tuple, will be interpreted as (h, m, s) or (d, m, s) depending on `unit`. If a string, it will be interpreted following the rules described above.

If `angle` is a sequence or array of strings, the resulting values will be in the given `unit`, or if `None` is provided, the unit will be taken from the first given value.

unit : `UnitBase`, str, optional

The unit of the value specified for the angle. This may be any string that `Unit` understands, but it is better to give an actual unit object. Must be an angular unit.

dtype : dtype, optional

See `Quantity`.

copy : bool, optional

See `Quantity`.

Raises

‘~astropy.units.UnitsError‘

If a unit is not provided or it is not an angular unit.

Attributes Summary

<code>dms</code>	The angle's value in degrees, as a named tuple with (d, m, s) members.
<code>hms</code>	The angle's value in hours, as a named tuple with (h, m, s) members.
<code>hour</code>	The angle's value in hours (read-only property).
<code>signed_dms</code>	The angle's value in degrees, as a named tuple with (sign, d, m, s) members.

Methods Summary

<code>format(*args, **kwargs)</code>	Deprecated since version 0.3.
<code>is_within_bounds([lower, upper])</code>	Check if all angle(s) satisfy <code>lower <= angle < upper</code> If <code>lower</code> is not specified, it is assumed to be 0.
<code>to_string([unit, decimal, sep, precision, ...])</code>	A string representation of the angle.

Table 10.5 – continued from previous page

`wrap_at(wrap_angle[, inplace])`Wrap the `Angle` object at the given `wrap_angle`.

Attributes Documentation

dms

The angle's value in degrees, as a named tuple with (`d`, `m`, `s`) members. (This is a read-only property.)

hms

The angle's value in hours, as a named tuple with (`h`, `m`, `s`) members. (This is a read-only property.)

hour

The angle's value in hours (read-only property).

signed_dms

The angle's value in degrees, as a named tuple with (`sign`, `d`, `m`, `s`) members. The `d`, `m`, `s` are thus always positive, and the sign of the angle is given by `sign`. (This is a read-only property.)

This is primarily intended for use with `dms` to generate string representations of coordinates that are correct for negative angles.

Methods Documentation

format (**args, **kwargs*)

Deprecated since version 0.3: The format function is deprecated and may be removed in a future version. Use `to_string` instead.

is_within_bounds (*lower=None, upper=None*)

Check if all angle(s) satisfy `lower <= angle < upper`

If `lower` is not specified (or `None`) then no lower bounds check is performed. Likewise `upper` can be left unspecified. For example:

```
>>> from astropy.coordinates import Angle
>>> import astropy.units as u
>>> a = Angle([-20, 150, 350] * u.deg)
>>> a.is_within_bounds('0d', '360d')
False
>>> a.is_within_bounds(None, '360d')
True
>>> a.is_within_bounds(-30 * u.deg, None)
True
```

Parameters

lower : str, `Angle`, angular `Quantity`, `None`

Specifies lower bound for checking. This can be any object that can initialize an `Angle` object, e.g. `'180d'`, `180 * u.deg`, or `Angle(180, unit=u.deg)`.

upper : str, `Angle`, angular `Quantity`, `None`

Specifies upper bound for checking. This can be any object that can initialize an `Angle` object, e.g. `'180d'`, `180 * u.deg`, or `Angle(180, unit=u.deg)`.

Returns

is_within_bounds : bool

`True` if all angles satisfy `lower <= angle < upper`

`to_string` (*unit=None, decimal=False, sep=u'fromunit', precision=None, alwayssign=False, pad=False, fields=3, format=None*)

A string representation of the angle.

Parameters

unit : `UnitBase`, optional

Specifies the unit. Must be an angular unit. If not provided, the unit used to initialize the angle will be used.

decimal : bool, optional

If `True`, a decimal representation will be used, otherwise the returned string will be in sexagesimal form.

sep : str, optional

The separator between numbers in a sexagesimal representation. E.g., if it is `'.'`, the result is `'12:41:11.1241'`. Also accepts 2 or 3 separators. E.g., `sep='hms'` would give the result `'12h41m11.1241s'`, or `sep='-:'` would yield `'11-21:17.124'`. Alternatively, the special string `'fromunit'` means `'dms'` if the unit is degrees, or `'hms'` if the unit is hours.

precision : int, optional

The level of decimal precision. If `decimal` is `True`, this is the raw precision, otherwise it gives the precision of the last place of the sexagesimal representation (seconds). If `None`, or not provided, the number of decimal places is determined by the value, and will be between 0-8 decimal places as required.

alwayssign : bool, optional

If `True`, include the sign no matter what. If `False`, only include the sign if it is negative.

pad : bool, optional

If `True`, include leading zeros when needed to ensure a fixed number of characters for sexagesimal representation.

fields : int, optional

Specifies the number of fields to display when outputting sexagesimal notation. For example:

- `fields == 1: '5d'`
- `fields == 2: '5d45m'`
- `fields == 3: '5d45m32.5s'`

By default, all fields are displayed.

format : str, optional

The format of the result. If not provided, an unadorned string is returned. Supported values are:

- `'latex'`: Return a LaTeX-formatted string
- `'unicode'`: Return a string containing non-ASCII unicode characters, such as the degree symbol

Returns

strrepr : str

A string representation of the angle.

`wrap_at` (*wrap_angle*, *inplace=False*)

Wrap the `Angle` object at the given `wrap_angle`.

This method forces all the angle values to be within a contiguous 360 degree range so that `wrap_angle - 360d <= angle < wrap_angle`. By default a new `Angle` object is returned, but if the `inplace` argument is `True` then the `Angle` object is wrapped in place and nothing is returned.

For instance:

```
>>> from astropy.coordinates import Angle
>>> import astropy.units as u
>>> a = Angle([-20.0, 150.0, 350.0] * u.deg)

>>> a.wrap_at(360 * u.deg).degree # Wrap into range 0 to 360 degrees
array([ 340., 150., 350.])

>>> a.wrap_at('180d', inplace=True) # Wrap into range -180 to 180 degrees
>>> a.degree
array([-20., 150., -10.]
```

Parameters

wrap_angle : str, `Angle`, angular `Quantity`

Specifies a single value for the wrap angle. This can be any object that can initialize an `Angle` object, e.g. `'180d'`, `180 * u.deg`, or `Angle(180, unit=u.deg)`.

inplace : bool

If `True` then wrap the object in place instead of returning a new `Angle`

Returns

out : `Angle` or `None`

If `inplace` is `False` (default), return new `Angle` object with angles wrapped accordingly. Otherwise wrap in place and return `None`.

BaseCoordinateFrame

`class astropy.coordinates.BaseCoordinateFrame (*args, **kwargs)`

Bases: `object`

The base class for coordinate frames.

This class is intended to be subclassed to create instances of specific systems. Subclasses can implement the following attributes:

- **default_representation**

A subclass of `BaseRepresentation` that will be treated as the default representation of this frame. This is the representation assumed by default when the frame is created.

- **FrameAttribute class attributes**

Frame attributes such as `FK4.equinox` or `FK4.obstime` are defined using a descriptor class. See the narrative documentation or built-in classes code for details.

- **frame_specific_representation_info**

A dictionary mapping the name or class of a representation to a list of `RepresentationMapping` objects that tell what names and default units should be used on this frame for the components of that representation.

Attributes Summary

<code>cartesian</code>	Shorthand for a cartesian representation of the coordinates in this object.
<code>data</code>	The coordinate data for this object.
<code>default_representation</code>	
<code>frame_specific_representation_info</code>	
<code>has_data</code>	True if this frame has <code>data</code> , False otherwise.
<code>isscalar</code>	
<code>name</code>	<code>str(object) -> string</code>
<code>representation</code>	The representation of the data in this frame, as a class that is subclassed from <code>BaseRepresentation</code> .
<code>representation_component_names</code>	
<code>representation_component_units</code>	
<code>representation_info</code>	A dictionary with the information of what attribute names for this frame apply to the data.
<code>shape</code>	
<code>spherical</code>	Shorthand for a spherical representation of the coordinates in this object.

Methods Summary

<code>get_frame_attr_names()</code>	
<code>is_frame_attr_default(attrnm)</code>	Determine whether or not a frame attribute has its value because it's the default value, or not.
<code>is_transformable_to(new_frame)</code>	Determines if this coordinate frame can be transformed to another given frame.
<code>realize_frame(representation)</code>	Generates a new frame <i>with new data</i> from another frame (which may or may not have data).
<code>represent_as(new_representation[, ...])</code>	Generate and return a new representation of this frame's <code>data</code> .
<code>separation(other)</code>	Computes on-sky separation between this coordinate and another.
<code>separation_3d(other)</code>	Computes three dimensional separation between this coordinate and another.
<code>transform_to(new_frame)</code>	Transform this object's coordinate data to a new frame.

Attributes Documentation

cartesian

Shorthand for a cartesian representation of the coordinates in this object.

data

The coordinate data for this object. If this frame has no data, an `ValueError` will be raised. Use `has_data` to check if data is present on this frame object.

default_representation

frame_specific_representation_info

has_data

True if this frame has `data`, False otherwise.

isscalar

name = 'basecoordinateframe'

representation

The representation of the data in this frame, as a class that is subclassed from `BaseRepresentation`. Can also be *set* using the string name of the representation.

representation_component_names

representation_component_units

representation_info

A dictionary with the information of what attribute names for this frame apply to particular representations.

shape

spherical

Shorthand for a spherical representation of the coordinates in this object.

Methods Documentation

classmethod `get_frame_attr_names()`

is_frame_attr_default (*attrnm*)

Determine whether or not a frame attribute has its value because it's the default value, or because this frame was created with that value explicitly requested.

Parameters

attrnm : str

The name of the attribute to check.

Returns

isdefault : bool

True if the attribute `attrnm` has its value by default, False if it was specified at creation of this frame.

is_transformable_to (*new_frame*)

Determines if this coordinate frame can be transformed to another given frame.

Parameters

new_frame : class or frame object

The proposed frame to transform into.

Returns

transformable : bool or str

True if this can be transformed to `new_frame`, False if not, or the string 'same' if `new_frame` is the same system as this object but no transformation is defined.

Notes

A return value of 'same' means the transformation will work, but it will just give back a copy of this object. The intended usage is:

```
if coord.is_transformable_to(some_unknown_frame):
    coord2 = coord.transform_to(some_unknown_frame)
```

This will work even if `some_unknown_frame` turns out to be the same frame class as `coord`. This is intended for cases where the frame is the same regardless of the frame attributes (e.g. ICRS), but be

aware that it *might* also indicate that someone forgot to define the transformation between two objects of the same frame class but with different attributes.

realize_frame (*representation*)

Generates a new frame *with new data* from another frame (which may or may not have data).

Parameters

representation : BaseRepresentation

The representation to use as the data for the new frame.

Returns

frameobj : same as this frame

A new object with the same frame attributes as this one, but with the *representation* as the data.

represent_as (*new_representation*, *in_frame_units=False*)

Generate and return a new representation of this frame's *data*.

Parameters

new_representation : subclass of BaseRepresentation or string

The type of representation to generate. May be a *class* (not an instance), or the string name of the representation class.

in_frame_units : bool

Force the representation units to match the specified units particular to this frame

Returns

newrep : whatever *new_representation* is

A new representation object of this frame's *data*.

Raises

AttributeError

If this object had no *data*

Examples

```
>>> from astropy import units as u
>>> from astropy.coordinates import ICRS, CartesianRepresentation
>>> coord = ICRS(0*u.deg, 0*u.deg)
>>> coord.represent_as(CartesianRepresentation)
<CartesianRepresentation x=1.0 , y=0.0 , z=0.0 >
```

separation (*other*)

Computes on-sky separation between this coordinate and another.

Parameters

other : BaseCoordinateFrame

The coordinate to get the separation to.

Returns

sep : Angle

The on-sky separation between this and the *other* coordinate.

Notes

The separation is calculated using the Vincenty formula, which is stable at all locations, including poles and antipodes [R2].

`separation_3d` (*other*)

Computes three dimensional separation between this coordinate and another.

Parameters

other : `BaseCoordinateFrame`

The coordinate system to get the distance to.

Returns

sep : `Distance`

The real-space distance between these two coordinates.

Raises

ValueError

If this or the other coordinate do not have distances.

`transform_to` (*new_frame*)

Transform this object's coordinate data to a new frame.

Parameters

new_frame : class or frame object or `SkyCoord` object

The frame to transform this coordinate frame into.

Returns

transformed

A new object with the coordinate data represented in the `newframe` system.

Raises

ValueError

If there is no possible transformation route.

BaseRepresentation

class `astropy.coordinates.BaseRepresentation`

Bases: `object`

Base Representation object, for representing a point in a 3D coordinate system.

Notes

All representation classes should subclass this base representation class. All subclasses should then define a `to_cartesian` method and a `from_cartesian` class method. By default, transformations are done via the cartesian system, but classes that want to define a smarter transformation path can overload the `represent_as` method. Furthermore, all classes must define an `attr_classes` attribute, an `OrderedDict` which maps component names to the class that creates them. They can also define a `recommended_units` dictionary, which maps component names to the units they are best presented to users in. Note that frame classes may override this with their own preferred units.

Attributes Summary

<code>components</code>	A tuple with the in-order names of the coordinate components
<code>isscalar</code>	
<code>recommended_units</code>	
<code>shape</code>	

Methods Summary

```
from_cartesian()  
from_representation(representation)  
get_name()  
represent_as(other_class)  
to_cartesian()
```

Attributes Documentation

components

A tuple with the in-order names of the coordinate components

isscalar

recommended_units = {}

shape

Methods Documentation

from_cartesian()

classmethod from_representation (*representation*)

classmethod get_name ()

represent_as (*other_class*)

to_cartesian()

BoundsError

exception `astropy.coordinates.BoundsError`

Raised when an angle is outside of its user-specified bounds.

CartesianPoints

class `astropy.coordinates.CartesianPoints` (*args, **kwargs)

Bases: `astropy.coordinates.CartesianPoints`

Deprecated since version v0.4: The CartesianPoints class is deprecated and may be removed in a future version. Use `astropy.coordinates.CartesianRepresentation` instead.

A cartesian representation of a point in three-dimensional space.

Parameters

x : `Quantity` or array-like

The first cartesian coordinate or a single array or `Quantity` where the first dimension is length-3.

y : `Quantity` or array-like, optional

The second cartesian coordinate.

z : `Quantity` or array-like, optional

The third cartesian coordinate.

unit : `UnitBase` object or `None`

The physical unit of the coordinate values. If `x`, `y`, or `z` are quantities, they will be converted to this unit.

dtype : `dtype`, optional

See `Quantity`. Must be given as a keyword argument.

copy : `bool`, optional

See `Quantity`. Must be given as a keyword argument.

Raises

UnitsError

If the units on `x`, `y`, and `z` do not match or an invalid unit is given.

ValueError

If `y` and `z` don't match `x`'s shape or `x` is not length-3

TypeError

If incompatible array types are passed into `x`, `y`, or `z`

Deprecated since version v0.4: The CartesianPoints class is deprecated and may be removed in a future version. Use `astropy.coordinates.CartesianRepresentation` instead.

CartesianRepresentation

class `astropy.coordinates.CartesianRepresentation` (x, y=None, z=None, copy=True)

Bases: `astropy.coordinates.BaseRepresentation`

Representation of points in 3D cartesian coordinates.

Parameters

x, y, z : `Quantity`

The `x`, `y`, and `z` coordinates of the point(s). If `x`, `y`, and `z` have different shapes, they should be broadcastable.

copy : bool, optional

If True arrays will be copied rather than referenced.

Attributes Summary

<code>attr_classes</code>	Dictionary that remembers insertion order
<code>x</code>	The x component of the point(s).
<code>xyz</code>	
<code>y</code>	The y component of the point(s).
<code>z</code>	The z component of the point(s).

Methods Summary

<code>from_cartesian(other)</code>
<code>to_cartesian()</code>

Attributes Documentation

attr_classes = `OrderedDict([(u'x', <class 'astropy.units.quantity.Quantity'>), (u'y', <class 'astropy.units.quantity.Qu`

x
The x component of the point(s).

xyz

y
The y component of the point(s).

z
The z component of the point(s).

Methods Documentation

classmethod `from_cartesian` (*other*)

`to_cartesian` ()

CompositeTransform

class `astropy.coordinates.CompositeTransform` (*transforms, fromsys, tosys, priority=1, register_graph=None, collapse_static_mats=True*)
Bases: `astropy.coordinates.CoordinateTransform`

A transformation constructed by combining together a series of single-step transformations.

Note that the intermediate frame objects are constructed using any frame attributes in `toframe` or `fromframe` that overlap with the intermediate frame (`toframe` favored over `fromframe` if there's a conflict). Any frame attributes that are not present use the defaults.

Parameters

transforms : sequence of `CoordinateTransform` objects

The sequence of transformations to apply.

fromsys : class

The coordinate frame class to start from.

tosys : class

The coordinate frame class to transform into.

priority : number

The priority if this transform when finding the shortest coordinate transform path - large numbers are lower priorities.

register_graph : `TransformGraph` or `None`

A graph to register this transformation with on creation, or `None` to leave it unregistered.

collapse_static_mats : bool

If `True`, consecutive `StaticMatrixTransform` will be collapsed into a single transformation to speed up the calculation.

Methods Summary

`__call__(fromcoord, toframe)`

Methods Documentation

`__call__(fromcoord, toframe)`

ConvertError

exception `astropy.coordinates.ConvertError`

Raised if a coordinate system cannot be converted to another

CoordinateTransform

class `astropy.coordinates.CoordinateTransform`(*fromsys*, *tosys*, *priority=1*, *register_graph=None*)

Bases: `object`

An object that transforms a coordinate from one system to another. Subclasses must implement `__call__` with the provided signature. They should also call this superclass's `__init__` in their `__init__`.

Parameters

fromsys : class

The coordinate frame class to start from.

tosys : class

The coordinate frame class to transform into.

priority : number

The priority if this transform when finding the shortest coordinate transform path - large numbers are lower priorities.

register_graph : `TransformGraph` or `None`

A graph to register this transformation with on creation, or `None` to leave it unregistered.

Methods Summary

<code>__call__(fromcoord, toframe)</code>	Does the actual coordinate transformation from the <code>fromsys</code> class to the <code>tosys</code> class.
<code>register(graph)</code>	Add this transformation to the requested Transformation graph, replacing anything already connected.
<code>unregister(graph)</code>	Remove this transformation from the requested transformation graph.

Methods Documentation

`__call__(fromcoord, toframe)`

Does the actual coordinate transformation from the `fromsys` class to the `tosys` class.

Parameters

fromcoord : `fromsys` object

An object of class matching `fromsys` that is to be transformed.

toframe : object

An object that has the attributes necessary to fully specify the frame. That is, it must have attributes with names that match the keys of the dictionary that `tosys.get_frame_attr_names()` returns. Typically this is of class `tosys`, but it *might* be some other class as long as it has the appropriate attributes.

Returns

tocoord : `tosys` object

The new coordinate after the transform has been applied.

`register(graph)`

Add this transformation to the requested Transformation graph, replacing anything already connecting these two coordinates.

Parameters

graph : a `TransformGraph` object

The graph to register this transformation with.

`unregister(graph)`

Remove this transformation from the requested transformation graph.

Parameters

graph : a `TransformGraph` object

The graph to unregister this transformation from.

Raises**ValueError**

If this is not currently in the transform graph.

CylindricalRepresentation

class `astropy.coordinates.CylindricalRepresentation` (*rho*, *phi*, *z*, *copy=True*)
 Bases: `astropy.coordinates.BaseRepresentation`

Representation of points in 3D cylindrical coordinates.

Parameters

rho : `Quantity`

The distance from the z axis to the point(s).

phi : `Quantity`

The azimuth of the point(s), in angular units, which will be wrapped to an angle between 0 and 360 degrees. This can also be instances of `Angle`,

z : `Quantity`

The z coordinate(s) of the point(s)

copy : bool, optional

If True arrays will be copied rather than referenced.

Attributes Summary

<code>attr_classes</code>	Dictionary that remembers insertion order
<code>phi</code>	The azimuth of the point(s).
<code>recommended_units</code>	
<code>rho</code>	The distance of the point(s) from the z-axis.
<code>z</code>	The height of the point(s).

Methods Summary

<code>from_cartesian(cart)</code>	Converts 3D rectangular cartesian coordinates to cylindrical polar coordinates.
<code>to_cartesian()</code>	Converts cylindrical polar coordinates to 3D rectangular cartesian coordinates.

Attributes Documentation

attr_classes = `OrderedDict([(u'rho', <class 'astropy.units.quantity.Quantity'>), (u'phi', <class 'astropy.coordinates.a`

phi

The azimuth of the point(s).

recommended_units = `{u'phi': Unit("deg")}`

rho

The distance of the point(s) from the z-axis.

z

The height of the point(s).

Methods Documentation

classmethod `from_cartesian` (*cart*)

Converts 3D rectangular cartesian coordinates to cylindrical polar coordinates.

to_cartesian ()

Converts cylindrical polar coordinates to 3D rectangular cartesian coordinates.

Distance

class `astropy.coordinates.Distance`Bases: `astropy.units.quantity.Quantity`

A one-dimensional distance.

This can be initialized in one of four ways:

- A distance `value` (array or float) and a unit
- A `Quantity` object
- A redshift and (optionally) a cosmology.
- Providing a distance modulus

Parameters

value : scalar or `Quantity`.

The value of this distance.

unit : `UnitBase`The units for this distance, *if* `value` is not a `Quantity`. Must have dimensions of distance.**z** : floatA redshift for this distance. It will be converted to a distance by computing the luminosity distance for this redshift given the cosmology specified by `cosmology`. Must be given as a keyword argument.**cosmology** : `Cosmology` or `None`A cosmology that will be used to compute the distance from `z`. If `None`, the current cosmology will be used (see `astropy.cosmology` for details).**distmod** : float or `Quantity`The distance modulus for this distance. Note that if `unit` is not provided, a guess will be made at the unit between AU, pc, kpc, and Mpc.**dtype** : `dtype`, optionalSee `Quantity`.**copy** : bool, optionalSee `Quantity`.**order** : {'C', 'F', 'A'}, optional

See `Quantity`.

subok : bool, optional

See `Quantity`.

ndmin : int, optional

See `Quantity`.

allow_negative : bool, optional

Whether to allow negative distances (which are possible in some cosmologies). Default: `False`.

Raises

‘~`astropy.units.UnitsError`‘

If the unit is not a distance.

ValueError

If value specified is less than 0 and `allow_negative=False`.

If `z` is provided with a unit or `cosmology` is provided when `z` is *not* given, or value is given as well as `z`.

Examples

```
>>> from astropy import units as u
>>> from astropy import cosmology
>>> from astropy.cosmology import WMAP5, WMAP7
>>> cosmology.set_current(WMAP7)
>>> d1 = Distance(10, u.Mpc)
>>> d2 = Distance(40, unit=u.au)
>>> d3 = Distance(value=5, unit=u.kpc)
>>> d4 = Distance(z=0.23)
>>> d5 = Distance(z=0.23, cosmology=WMAP5)
>>> d6 = Distance(distmod=24.47)
>>> d7 = Distance(Distance(10 * u.Mpc))
```

Attributes Summary

<code>distmod</code>	The distance modulus as a <code>Quantity</code>
<code>z</code>	Short for <code>self.compute_z()</code>

Methods Summary

<code>compute_z([cosmology])</code>	The redshift for this distance assuming its physical distance is a luminosity distance.
-------------------------------------	---

Attributes Documentation

`distmod`

The distance modulus as a `Quantity`

`z`

Short for `self.compute_z()`

Methods Documentation

compute_z (*cosmology=None*)

The redshift for this distance assuming its physical distance is a luminosity distance.

Parameters

cosmology : `Cosmology` or `None`

The cosmology to assume for this calculation, or `None` to use the current cosmology (see `astropy.cosmology` for details).

Returns

z : float

The redshift of this distance given the provided `cosmology`.

DynamicMatrixTransform

class `astropy.coordinates.DynamicMatrixTransform` (*matrix_func, fromsys, tosys, priority=1, register_graph=None*)

Bases: `astropy.coordinates.CoordinateTransform`

A coordinate transformation specified as a function that yields a 3 x 3 cartesian transformation matrix.

This is similar to, but distinct from `StaticMatrixTransform`, in that the matrix for this class might depend on frame attributes.

Parameters

matrix_func : callable

A callable that has the signature `matrix_func(fromcoord, toframe)` and returns a 3 x 3 matrix that converts `fromcoord` in a cartesian representation to the new coordinate system.

fromsys : class

The coordinate frame class to start from.

tosys : class

The coordinate frame class to transform into.

priority : number

The priority if this transform when finding the shortest coordinate transform path - large numbers are lower priorities.

register_graph : `TransformGraph` or `None`

A graph to register this transformation with on creation, or `None` to leave it unregistered.

Raises

TypeError

If `matrix_func` is not callable

Methods Summary

`__call__(fromcoord, toframe)`

Methods Documentation

`__call__(fromcoord, toframe)`

EarthLocation

class `astropy.coordinates.EarthLocation`

Bases: `astropy.units.quantity.Quantity`

Location on Earth.

Initialization is first attempted assuming geocentric (x, y, z) coordinates are given; if that fails, another attempt is made assuming geodetic coordinates (longitude, latitude, height above a reference ellipsoid). Internally, the coordinates are stored as geocentric.

To ensure a specific type of coordinates is used, use the corresponding class methods (`from_geocentric` and `from_geodetic`) or initialize the arguments with names (x, y, z for geocentric; lon, lat, height for geodetic). See the class methods for details.

Notes

For conversion to and from geodetic coordinates, the ERFA routines `gc2gd` and `gd2gc` are used. See <https://github.com/liberfa/erfa>

Attributes Summary

<code>ellipsoid</code>	The default ellipsoid used to convert to geodetic coordinates.
<code>geocentric</code>	Convert to a tuple with X, Y, and Z as quantities
<code>geodetic</code>	Convert to geodetic coordinates for the default ellipsoid.
<code>height</code>	Height of the location, for the default ellipsoid.
<code>latitude</code>	Latitude of the location, for the default ellipsoid.
<code>longitude</code>	Longitude of the location, for the default ellipsoid.
<code>x</code>	The X component of the geocentric coordinates.
<code>y</code>	The Y component of the geocentric coordinates.
<code>z</code>	The Z component of the geocentric coordinates.

Methods Summary

<code>from_geocentric(x, y, z[, unit])</code>	Location on Earth, initialized from geocentric coordinates.
<code>from_geodetic(lon, lat[, height, ellipsoid])</code>	Location on Earth, initialized from geodetic coordinates.
<code>to(unit[, equivalencies])</code>	Returns a new <code>Quantity</code> object with the specified units.
<code>to_geocentric()</code>	Convert to a tuple with X, Y, and Z as quantities
<code>to_geodetic([ellipsoid])</code>	Convert to geodetic coordinates.

Attributes Documentation

ellipsoid

The default ellipsoid used to convert to geodetic coordinates.

geocentric

Convert to a tuple with X, Y, and Z as quantities

geodetic

Convert to geodetic coordinates for the default ellipsoid.

height

Height of the location, for the default ellipsoid.

latitude

Latitude of the location, for the default ellipsoid.

longitude

Longitude of the location, for the default ellipsoid.

x

The X component of the geocentric coordinates.

y

The Y component of the geocentric coordinates.

z

The Z component of the geocentric coordinates.

Methods Documentation

classmethod from_geocentric (*x*, *y*, *z*, *unit=None*)

Location on Earth, initialized from geocentric coordinates.

Parameters

x, y, z : `Quantity` or array-like

Cartesian coordinates. If not quantities, `unit` should be given.

unit : `UnitBase` object or None

Physical unit of the coordinate values. If `x`, `y`, and/or `z` are quantities, they will be converted to this unit.

Raises

astropy.units.UnitsError

If the units on `x`, `y`, and `z` do not match or an invalid unit is given.

ValueError

If the shapes of `x`, `y`, and `z` do not match.

TypeError

If `x` is not a `Quantity` and no unit is given.

classmethod from_geodetic (*lon*, *lat*, *height=0.0*, *ellipsoid=None*)

Location on Earth, initialized from geodetic coordinates.

Parameters

lon : `Longitude` or float

Earth East longitude. Can be anything that initialises an `Angle` object (if float, in degrees).

lat : `Latitude` or float

Earth latitude. Can be anything that initialises an `Latitude` object (if float, in degrees).

height : `Quantity` or float, optional

Height above reference ellipsoid (if float, in meters; default: 0).

ellipsoid : str, optional

Name of the reference ellipsoid to use (default: 'WGS84'). Available ellipsoids are: 'WGS84', 'GRS80', 'WGS72'.

Raises

astropy.units.UnitsError

If the units on `lon` and `lat` are inconsistent with angular ones, or that on `height` with a length.

ValueError

If `lon`, `lat`, and `height` do not have the same shape, or if `ellipsoid` is not recognized as among the ones implemented.

Notes

For the conversion to geocentric coordinates, the ERFA routine `gd2gc` is used. See <https://github.com/liberfa/erfa>

to (*unit, equivalencies=[]*)

Returns a new `Quantity` object with the specified units.

Parameters

unit : `UnitBase` instance, str

An object that represents the unit to convert to. Must be an `UnitBase` object or a string parseable by the `units` package.

equivalencies : list of equivalence pairs, optional

A list of equivalence pairs to try if the units are not directly convertible. See *Equivalencies*. If not provided or [], class default equivalencies will be used (none for `Quantity`, but may be set for subclasses) If `None`, no equivalencies will be applied at all, not even any set globally or within a context.

to_geocentric ()

Convert to a tuple with X, Y, and Z as quantities

to_geodetic (*ellipsoid=None*)

Convert to geodetic coordinates.

Parameters

ellipsoid : str, optional

Reference ellipsoid to use. Default is the one the coordinates were initialized with. Available are: 'WGS84', 'GRS80', 'WGS72'

Returns

(**lon, lat, height**) : tuple

The tuple contains instances of `Longitude`, `Latitude`, and `Quantity`

Raises

ValueError

if `ellipsoid` is not recognized as among the ones implemented.

Notes

For the conversion to geodetic coordinates, the ERFA routine `gc2gd` is used. See <https://github.com/liberfa/erfa>

FK4

```
class astropy.coordinates.FK4 (*args, **kwargs)
    Bases: astropy.coordinates.BaseCoordinateFrame
```

A coordinate or frame in the FK4 system.

Parameters

representation : `BaseRepresentation` or `None`

A representation object or `None` to have no data (or use the other keywords)

ra : `Angle`, optional, must be keyword

The RA for this object (`dec` must also be given and `representation` must be `None`).

dec : `Angle`, optional, must be keyword

The Declination for this object (`ra` must also be given and `representation` must be `None`).

distance : `Quantity`, optional, must be keyword

The Distance for this object along the line-of-sight. (`representation` must be `None`).

equinox : `astropy.time.Time`, optional, must be keyword

The equinox of this frame.

obstime : `astropy.time.Time`, optional, must be keyword

The time this frame was observed. If `None`, will be the same as `equinox`.

Attributes Summary

<code>default_representation</code>	
<code>equinox</code>	Represent and manipulate times and dates for astronomy.
<code>frame_specific_representation_info</code>	
<code>name</code>	<code>str(object) -> string</code>
<code>obstime</code>	

Attributes Documentation

`default_representation`

`equinox = <Time object: scale='tai' format='byear_str' value=B1950.000>`

`frame_specific_representation_info`

`name = 'fk4'`

`obstime = None`

FK4NoETerms

`class astropy.coordinates.FK4NoETerms (*args, **kwargs)`

Bases: `astropy.coordinates.BaseCoordinateFrame`

A coordinate or frame in the FK4 system, but with the E-terms of aberration removed.

Parameters

representation : `BaseRepresentation` or `None`

A representation object or `None` to have no data (or use the other keywords)

ra : `Angle`, optional, must be keyword

The RA for this object (`dec` must also be given and `representation` must be `None`).

dec : `Angle`, optional, must be keyword

The Declination for this object (`ra` must also be given and `representation` must be `None`).

distance : `Quantity`, optional, must be keyword

The Distance for this object along the line-of-sight. (`representation` must be `None`).

obstime : `astropy.time.Time`, optional, must be keyword

The time this frame was observed. If `None`, will be the same as `equinox`.

Attributes Summary

<code>default_representation</code>	
<code>equinox</code>	Represent and manipulate times and dates for astronomy.
<code>frame_specific_representation_info</code>	
<code>name</code>	<code>str(object) -> string</code>
<code>obstime</code>	

Attributes Documentation

`default_representation`

`equinox = <Time object: scale='tai' format='byear_str' value=B1950.000>`

`frame_specific_representation_info`

`name = 'fk4noeterns'`

`obstime = None`

FK5

class `astropy.coordinates.FK5` (**args*, ***kwargs*)

Bases: `astropy.coordinates.BaseCoordinateFrame`

A coordinate or frame in the FK5 system.

Parameters

representation : `BaseRepresentation` or `None`

A representation object or `None` to have no data (or use the other keywords)

ra : `Angle`, optional, must be keyword

The RA for this object (dec must also be given and `representation` must be `None`).

dec : `Angle`, optional, must be keyword

The Declination for this object (`ra` must also be given and `representation` must be `None`).

distance : `Quantity`, optional, must be keyword

The Distance for this object along the line-of-sight. (`representation` must be `None`).

equinox : `Time`, optional, must be keyword

The equinox of this frame.

Attributes Summary

<code>default_representation</code>	
<code>equinox</code>	Represent and manipulate times and dates for astronomy.
<code>frame_specific_representation_info</code>	
<code>name</code>	<code>str(object) -> string</code>

Attributes Documentation

`default_representation`

`equinox = <Time object: scale='utc' format='jyear_str' value=J2000.000>`

`frame_specific_representation_info`

`name = 'fk5'`

FrameAttribute

`class astropy.coordinates.FrameAttribute (default=None, secondary_attribute='')`

Bases: `object`

A non-mutable data descriptor to hold a frame attribute.

This class must be used to define frame attributes (e.g. `equinox` or `obstime`) that are included in a frame class definition.

Parameters

default : object

Default value for the attribute if not provided

secondary_attribute : str

Name of a secondary instance attribute which supplies the value if `default` is `None` and no value was supplied during initialization.

Returns

frame_attr : descriptor

A new data descriptor to hold a frame attribute

Examples

The `FK4` class uses the following class attributes:

```
class FK4(BaseCoordinateFrame):
    equinox = TimeFrameAttribute(default=_EQUINOX_B1950)
    obstime = TimeFrameAttribute(default=None, secondary_attribute='equinox')
```

This means that `equinox` and `obstime` are available to be set as keyword arguments when creating an `FK4` class instance and are then accessible as instance attributes. The instance value for the attribute must be stored in `'_' + <attribute_name>` by the frame `__init__` method.

Note in this example that `equinox` and `obstime` are time attributes and use the `TimeAttributeFrame` class. This subclass overrides the `convert_input` method to validate and convert inputs into a `Time` object.

Methods Summary

`convert_input(value)` Validate the input `value` and convert to expected attribute class.

Methods Documentation

`convert_input` (*value*)

Validate the input `value` and convert to expected attribute class.

The base method here does nothing, but subclasses can implement this as needed. The method should catch any internal exceptions and raise `ValueError` with an informative message.

The method returns the validated input along with a boolean that indicates whether the input value was actually converted. If the input value was already the correct type then the `converted` return value should be `False`.

Parameters

value : object

Input value to be converted.

Returns

`output_value`

The `value` converted to the correct type (or just `value` if `converted` is `False`)

converted : bool

True if the conversion was actually performed, `False` otherwise.

Raises

ValueError

If the input is not valid for this attribute.

FunctionTransform

class `astropy.coordinates.FunctionTransform` (*func*, *fromsys*, *tosys*, *priority=1*, *register_graph=None*)

Bases: `astropy.coordinates.CoordinateTransform`

A coordinate transformation defined by a function that accepts a coordinate object and returns the transformed coordinate object.

Parameters

func : callable

The transformation function. Should have a call signature `func(formcoord, toframe)`. Note that, unlike `CoordinateTransform.__call__`, `toframe` is assumed to be of type `tosys` for this function.

fromsys : class

The coordinate frame class to start from.

tosys : class

The coordinate frame class to transform into.

priority : number

The priority if this transform when finding the shortest coordinate transform path - large numbers are lower priorities.

register_graph : `TransformGraph` or `None`

A graph to register this transformation with on creation, or `None` to leave it unregistered.

Raises

TypeError

If `func` is not callable.

ValueError

If `func` cannot accept two arguments.

Methods Summary

`__call__(fromcoord, toframe)`

Methods Documentation

`__call__(fromcoord, toframe)`

Galactic

class `astropy.coordinates.Galactic` (**args, **kwargs*)
Bases: `astropy.coordinates.BaseCoordinateFrame`

Galactic Coordinates.

Parameters

representation : `BaseRepresentation` or `None`

A representation object or `None` to have no data (or use the other keywords)

l : `Angle`, optional, must be keyword

The Galactic longitude for this object (`b` must also be given and `representation` must be `None`).

b : `Angle`, optional, must be keyword

The Galactic latitude for this object (`l` must also be given and `representation` must be `None`).

distance : `Quantity`, optional, must be keyword

The Distance for this object along the line-of-sight.

Attributes Summary

<code>default_representation</code>	
<code>frame_specific_representation_info</code>	
<code>name</code>	<code>str(object) -> string</code>

Attributes Documentation`default_representation``frame_specific_representation_info``name = 'galactic'`**GenericFrame**

class `astropy.coordinates.GenericFrame` (*frame_attrs*)
 Bases: `astropy.coordinates.BaseCoordinateFrame`

A frame object that can't store data but can hold any arbitrary frame attributes. Mostly useful as a utility for the high-level class to store intermediate frame attributes.

Parameters**frame_attrs** : dict

A dictionary of attributes to be used as the frame attributes for this frame.

Attributes Summary

```
default_representation
frame_specific_representation_info
name
```

Methods Summary

```
get_frame_attr_names()
```

Attributes Documentation`default_representation``frame_specific_representation_info``name = None`**Methods Documentation**`get_frame_attr_names()`

ICRS

class `astropy.coordinates.ICRS` (**args*, ***kwargs*)
Bases: `astropy.coordinates.BaseCoordinateFrame`

A coordinate or frame in the ICRS system.

If you're looking for "J2000" coordinates, and aren't sure if you want to use this or `FK5`, you probably want to use ICRS. It's more well-defined as a catalog coordinate and is an inertial system, and is very close (within tens of arcseconds) to J2000 equatorial.

Parameters

representation : `BaseRepresentation` or `None`

A representation object or `None` to have no data (or use the other keywords)

ra : `Angle`, optional, must be keyword

The RA for this object (`dec` must also be given and `representation` must be `None`).

dec : `Angle`, optional, must be keyword

The Declination for this object (`ra` must also be given and `representation` must be `None`).

distance : `Quantity`, optional, must be keyword

The Distance for this object along the line-of-sight. (`representation` must be `None`).

Attributes Summary

<code>default_representation</code>	
<code>frame_specific_representation_info</code>	
<code>name</code>	<code>str(object) -> string</code>

Attributes Documentation

default_representation

frame_specific_representation_info

name = 'icrs'

IllegalHourError

exception `astropy.coordinates.IllegalHourError` (*hour*)
Raised when an hour value is not in the range [0,24).

Parameters

hour : int, float

Examples

```
if not 0 <= hr < 24:
    raise IllegalHourError(hour)
```

IllegalHourWarning

exception `astropy.coordinates.IllegalHourWarning` (*hour*, *alternativeactionstr=None*)
 Raised when an hour value is 24.

Parameters

hour : int, float

IllegalMinuteError

exception `astropy.coordinates.IllegalMinuteError` (*minute*)
 Raised when an minute value is not in the range [0,60].

Parameters

minute : int, float

Examples

```
if not 0 <= min < 60:
    raise IllegalMinuteError(minute)
```

IllegalMinuteWarning

exception `astropy.coordinates.IllegalMinuteWarning` (*minute*, *alternativeactionstr=None*)
 Raised when a minute value is 60.

Parameters

minute : int, float

IllegalSecondError

exception `astropy.coordinates.IllegalSecondError` (*second*)
 Raised when an second value (time) is not in the range [0,60].

Parameters

second : int, float

Examples

```
if not 0 <= sec < 60:
    raise IllegalSecondError(second)
```

IllegalSecondWarning

exception `astropy.coordinates.IllegalSecondWarning` (*second, alternativeactionstr=None*)
Raised when a second value is 60.

Parameters

second : int, float

Latitude

class `astropy.coordinates.Latitude`

Bases: `astropy.coordinates.Angle`

Latitude-like angle(s) which must be in the range -90 to +90 deg.

A Latitude object is distinguished from a pure `Angle` by virtue of being constrained so that:

```
-90.0 * u.deg <= angle(s) <= +90.0 * u.deg
```

Any attempt to set a value outside that range will result in a `ValueError`.

The input angle(s) can be specified either as an array, list, scalar, tuple (see below), string, `Quantity` or another `Angle`.

The input parser is flexible and supports all of the input formats supported by `Angle`.

Parameters

angle : array, list, scalar, `Quantity`, `Angle`. The

angle value(s). If a tuple, will be interpreted as (h, m, s) or (d, m, s) depending on `unit`. If a string, it will be interpreted following the rules described for `Angle`.

If `angle` is a sequence or array of strings, the resulting values will be in the given `unit`, or if `None` is provided, the unit will be taken from the first given value.

unit : `UnitBase`, str, optional

The unit of the value specified for the angle. This may be any string that `Unit` understands, but it is better to give an actual unit object. Must be an angular unit.

Raises

‘~`astropy.units.UnitsError`‘

If a unit is not provided or it is not an angular unit.

‘`TypeError`‘

If the angle parameter is an instance of `Longitude`.

Longitude

class `astropy.coordinates.Longitude`

Bases: `astropy.coordinates.Angle`

Longitude-like angle(s) which are wrapped within a contiguous 360 degree range.

A `Longitude` object is distinguished from a pure `Angle` by virtue of a `wrap_angle` property. The `wrap_angle` specifies that all angle values represented by the object will be in the range:

```
wrap_angle - 360 * u.deg <= angle(s) < wrap_angle
```

The default `wrap_angle` is 360 deg. Setting `wrap_angle=180 * u.deg` would instead result in values between -180 and +180 deg. Setting the `wrap_angle` attribute of an existing `Longitude` object will result in re-wrapping the angle values in-place.

The input angle(s) can be specified either as an array, list, scalar, tuple, string, `Quantity` or another `Angle`.

The input parser is flexible and supports all of the input formats supported by `Angle`.

Parameters

angle : array, list, scalar, `Quantity`,

`Angle` The angle value(s). If a tuple, will be interpreted as (h, m s) or (d, m, s) depending on `unit`. If a string, it will be interpreted following the rules described for `Angle`.

If `angle` is a sequence or array of strings, the resulting values will be in the given `unit`, or if `None` is provided, the `unit` will be taken from the first given value.

unit : `UnitBase`, str, optional

The `unit` of the value specified for the angle. This may be any string that `Unit` understands, but it is better to give an actual `unit` object. Must be an angular `unit`.

wrap_angle : `Angle` or equivalent, or `None`

Angle at which to wrap back to `wrap_angle - 360 deg`. If `None` (default), it will be taken to be 360 deg unless `angle` has a `wrap_angle` attribute already (i.e., is a `Longitude`), in which case it will be taken from there.

Raises

‘~astropy.units.UnitsError‘

If a `unit` is not provided or it is not an angular `unit`.

‘TypeError‘

If the `angle` parameter is an instance of `Latitude`.

Attributes Summary

`wrap_angle`

Attributes Documentation

wrap_angle

PhysicsSphericalRepresentation

class `astropy.coordinates.PhysicsSphericalRepresentation` (*phi, theta, r, copy=True*)
 Bases: `astropy.coordinates.BaseRepresentation`

Representation of points in 3D spherical coordinates (using the physics convention of using `phi` and `theta` for azimuth and inclination from the pole).

Parameters

phi, theta : `Quantity` or str

The azimuth and inclination of the point(s), in angular units. The inclination should be between 0 and 180 degrees, and the azimuth will be wrapped to an angle between 0 and 360 degrees. These can also be instances of `Angle`. If `copy` is `False`, `phi` will be changed inplace if it is not between 0 and 360 degrees.

r : `Quantity`

The distance to the point(s). If the distance is a length, it is passed to the `Distance` class, otherwise it is passed to the `Quantity` class.

copy : bool, optional

If `True` arrays will be copied rather than referenced.

Attributes Summary

<code>attr_classes</code>	Dictionary that remembers insertion order
<code>phi</code>	The azimuth of the point(s).
<code>r</code>	The distance from the origin to the point(s).
<code>recommended_units</code>	
<code>theta</code>	The elevation of the point(s).

Methods Summary

<code>from_cartesian(cart)</code>	Converts 3D rectangular cartesian coordinates to spherical polar coordinates.
<code>represent_as(other_class)</code>	
<code>to_cartesian()</code>	Converts spherical polar coordinates to 3D rectangular cartesian coordinates.

Attributes Documentation

attr_classes = `OrderedDict([(u'phi', <class 'astropy.coordinates.angles.Angle'>), (u'theta', <class 'astropy.coordinates.angles.Angle'>)])`

phi

The azimuth of the point(s).

r

The distance from the origin to the point(s).

recommended_units = `{u'theta': Unit("deg"), u'phi': Unit("deg")}`

theta

The elevation of the point(s).

Methods Documentation

classmethod from_cartesian (*cart*)

Converts 3D rectangular cartesian coordinates to spherical polar coordinates.

represent_as (*other_class*)

`to_cartesian()`

Converts spherical polar coordinates to 3D rectangular cartesian coordinates.

RangeError

exception `astropy.coordinates.RangeError`

Raised when some part of an angle is out of its valid range.

RepresentationMapping

class `astropy.coordinates.RepresentationMapping`

Bases: `astropy.coordinates.RepresentationMapping`

This `namedtuple` is used with the `frame_specific_representation_info` attribute to tell frames what attribute names (and default units) to use for a particular representation. `reprname` and `framename` should be strings, while `defaultunit` can be either an astropy unit, the string 'recommended' (to use whatever the representation's `recommended_units` is), or `None` (to indicate that no unit mapping should be done).

SkyCoord

class `astropy.coordinates.SkyCoord(*args, **kwargs)`

Bases: `object`

High-level object providing a flexible interface for celestial coordinate representation, manipulation, and transformation between systems.

The `SkyCoord` class accepts a wide variety of inputs for initialization. At a minimum these must provide one or more celestial coordinate values with unambiguous units. Typically one also specifies the coordinate frame, though this is not required. The general pattern is for spherical representations is:

```
SkyCoord(COORD, [FRAME], keyword_args ...)
SkyCoord(LON, LAT, [FRAME], keyword_args ...)
SkyCoord(LON, LAT, [DISTANCE], frame=FRAME, unit=UNIT, keyword_args ...)
SkyCoord([FRAME], <lon_attr>=LON, <lat_attr>=LAT, keyword_args ...)
```

It is also possible to input coordinate values in other representations such as cartesian or cylindrical. In this case one includes the keyword argument `representation='cartesian'` (for example) along with data in `x`, `y`, and `z`.

Parameters

frame : `BaseCoordinateFrame` class or string, optional

Type of coordinate frame this `SkyCoord` should represent. Defaults to `ICRS` if not given or given as `None`.

unit : `Unit`, string, or tuple of `Unit` or str, optional

Units for supplied `LON` and `LAT` values, respectively. If only one unit is supplied then it applies to both `LON` and `LAT`.

obstime : valid `Time` initializer, optional

Time of observation

equinox : valid `Time` initializer, optional

Coordinate frame equinox

representation : str or Representation class

Specifies the representation, e.g. 'spherical', 'cartesian', or 'cylindrical'. This affects the positional args and other keyword args which must correspond to the given representation.

****keyword_args**

Other keyword arguments as applicable for user-defined coordinate frames. Common options include:

ra, dec

[valid [Angle](#) initializer, optional] RA and Dec for frames where ra and dec are keys in the frame's `representation_component_names`, including ICRS, FK5, FK4, and FK4NoETerms.

l, b

[valid [Angle](#) initializer, optional] Galactic l and b for for frames where l and b are keys in the frame's `representation_component_names`, including the Galactic frame.

x, y, z

[float or [Quantity](#), optional] Cartesian coordinates values

w, u, v

[float or [Quantity](#), optional] Cartesian coordinates values for the Galactic frame.

Examples

The examples below illustrate common ways of initializing a `SkyCoord` object. For a complete description of the allowed syntax see the full coordinates documentation. First some imports:

```
>>> from astropy.coordinates import SkyCoord # High-level coordinates
>>> from astropy.coordinates import ICRS, Galactic, FK4, FK5 # Low-level frames
>>> from astropy.coordinates import Angle, Latitude, Longitude # Angles
>>> import astropy.units as u
```

The coordinate values and frame specification can now be provided using positional and keyword arguments:

```
>>> c = SkyCoord(10, 20, unit="deg") # defaults to ICRS frame
>>> c = SkyCoord([1, 2, 3], [-30, 45, 8], "icrs", unit="deg") # 3 coords

>>> coords = ["1:12:43.2 +1:12:43", "1 12 43.2 +1 12 43"]
>>> c = SkyCoord(coords, FK4, unit=(u.deg, u.hourangle), obstime="J1992.21")

>>> c = SkyCoord("1h12m43.2s +1d12m43s", Galactic) # Units from string
>>> c = SkyCoord("galactic", l="1h12m43.2s", b="+1d12m43s")

>>> ra = Longitude([1, 2, 3], unit=u.deg) # Could also use Angle
>>> dec = np.array([4.5, 5.2, 6.3]) * u.deg # Astropy Quantity
>>> c = SkyCoord(ra, dec, frame='icrs')
>>> c = SkyCoord(ICRS, ra=ra, dec=dec, obstime='2001-01-02T12:34:56')

>>> c = FK4(1 * u.deg, 2 * u.deg) # Uses defaults for obstime, equinox
>>> c = SkyCoord(c, obstime='J2010.11', equinox='B1965') # Override defaults

>>> c = SkyCoord(w=0, u=1, v=2, unit='kpc', frame='galactic', representation='cartesian')
```

As shown, the frame can be a `BaseCoordinateFrame` class or the corresponding string alias. The frame classes that are built in to astropy are `ICRS`, `FK5`, `FK4`, `FK4NoETerms`, and `Galactic`. The string aliases are simply lower-case versions of the class name, and allow for creating a `SkyCoord` object and transforming frames without explicitly importing the frame classes.

Attributes Summary

```
frame
representation
```

Methods Summary

<code>from_name(name[, frame])</code>	Given a name, query the CDS name resolver to attempt to retrieve coordinate information for that object.
<code>match_to_catalog_3d(catalogcoord[, nthneighbor])</code>	Finds the nearest 3-dimensional matches of this coordinate to a set of catalog coordinates.
<code>match_to_catalog_sky(catalogcoord[, nthneighbor])</code>	Finds the nearest on-sky matches of this coordinate in a set of catalog coordinates.
<code>position_angle(other)</code>	Computes the on-sky position angle (East of North) between this <code>SkyCoord</code> and another.
<code>separation(other)</code>	Computes on-sky separation between this coordinate and another.
<code>separation_3d(other)</code>	Computes three dimensional separation between this coordinate and another.
<code>to_string([style])</code>	A string representation of the coordinates.
<code>transform_to(frame)</code>	Transform this coordinate to a new frame.

Attributes Documentation

frame

representation

Methods Documentation

classmethod `from_name` (*name*, *frame=u'icrs'*)

Given a name, query the CDS name resolver to attempt to retrieve coordinate information for that object. The search database, sesame url, and query timeout can be set through configuration items in `astropy.coordinates.name_resolve` – see docstring for `get_icrs_coordinates` for more information.

Parameters

name : str

The name of the object to get coordinates for, e.g. 'M42'.

frame : str or `BaseCoordinateFrame` class or instance

The frame to transform the object to.

Returns

coord : `SkyCoord`

Instance of the `SkyCoord` class.

`match_to_catalog_3d` (*catalogcoord*, *nthneighbor=1*)

Finds the nearest 3-dimensional matches of this coordinate to a set of catalog coordinates.

This finds the 3-dimensional closest neighbor, which is only different from the on-sky distance if distance is set in this object or the `catalogcoord` object.

Parameters

catalogcoord : `SkyCoord` or `BaseCoordinateFrame`

The base catalog in which to search for matches. Typically this will be a coordinate object that is an array (i.e., `catalogcoord.isscalar == False`)

nthneighbor : int, optional

Which closest neighbor to search for. Typically 1 is desired here, as that is correct for matching one set of coordinates to another. The next likely use case is 2, for matching a coordinate catalog against *itself* (1 is inappropriate because each point will find itself as the closest match).

Returns

idx : integer array

Indices into `catalogcoord` to get the matched points for each of this object's coordinates. Shape matches this object.

sep2d : `Angle`

The on-sky separation between the closest match for each element in this object in `catalogcoord`. Shape matches this object.

dist3d : `Quantity`

The 3D distance between the closest match for each element in this object in `catalogcoord`. Shape matches this object.

See also:

`astropy.coordinates.match_coordinates_3d`

Notes

This method requires `SciPy` to be installed or it will fail.

match_to_catalog_sky (*catalogcoord*, *nthneighbor=1*)

Finds the nearest on-sky matches of this coordinate in a set of catalog coordinates.

Parameters

catalogcoord : `SkyCoord` or `BaseCoordinateFrame`

The base catalog in which to search for matches. Typically this will be a coordinate object that is an array (i.e., `catalogcoord.isscalar == False`)

nthneighbor : int, optional

Which closest neighbor to search for. Typically 1 is desired here, as that is correct for matching one set of coordinates to another. The next likely use case is 2, for matching a coordinate catalog against *itself* (1 is inappropriate because each point will find itself as the closest match).

Returns

idx : integer array

Indices into `catalogcoord` to get the matched points for each of this object's coordinates. Shape matches this object.

sep2d : `Angle`

The on-sky separation between the closest match for each element in this object in `catalogcoord`. Shape matches this object.

dist3d : `Quantity`

The 3D distance between the closest match for each element in this object in `catalogcoord`. Shape matches this object.

See also:

`astropy.coordinates.match_coordinates_sky`

Notes

This method requires `SciPy` to be installed or it will fail.

position_angle (*other*)

Computes the on-sky position angle (East of North) between this `SkyCoord` and another.

Parameters

other : `SkyCoord`

The other coordinate to compute the position angle to. It is treated as the “head” of the vector of the position angle.

Returns

pa : `Angle`

The (positive) position angle of the vector pointing from `self` to `other`. If either `self` or `other` contain arrays, this will be an array following the appropriate `numpy` broadcasting rules.

Examples

```
>>> c1 = SkyCoord(0*u.deg, 0*u.deg)
>>> c2 = SkyCoord(1*u.deg, 0*u.deg)
>>> c1.position_angle(c2).degree
90.0
>>> c3 = SkyCoord(1*u.deg, 1*u.deg)
>>> c1.position_angle(c3).degree
44.995636455344844
```

separation (*other*)

Computes on-sky separation between this coordinate and another.

Parameters

other : `SkyCoord` or `BaseCoordinateFrame`

The coordinate to get the separation to.

Returns

sep : `Angle`

The on-sky separation between this and the `other` coordinate.

Notes

The separation is calculated using the Vincenty formula, which is stable at all locations, including poles and antipodes [R4].

separation_3d (*other*)

Computes three dimensional separation between this coordinate and another.

Parameters

other : `SkyCoord` or `BaseCoordinateFrame`

The coordinate to get the separation to.

Returns

sep : `Distance`

The real-space distance between these two coordinates.

Raises**ValueError**

If this or the other coordinate do not have distances.

to_string (*style=u'decimal', **kwargs*)

A string representation of the coordinates.

The default styles definitions are:

```
'decimal': {'lat': {'decimal': True, 'unit': "deg"},
            'lon': {'decimal': True, 'unit': "deg"}}
'dms': {'lat': {'unit': "deg"},
        'lon': {'unit': "deg"}}
'hmsdms': {'lat': {'alwayssign': True, 'pad': True, 'unit': "deg"},
           'lon': {'pad': True, 'unit': "hour"}}
```

See `to_string()` for details and keyword arguments (the two angles forming the coordinates are both `Angle` instances). Keyword arguments have precedence over the style defaults and are passed to `to_string()`.

Parameters

style : {'hmsdms', 'dms', 'decimal'}

The formatting specification to use. These encode the three most common ways to represent coordinates. The default is `decimal`.

kwargs

Keyword args passed to `to_string()`.

transform_to (*frame*)

Transform this coordinate to a new frame.

Parameters

frame : str or `BaseCoordinateFrame` class / instance or `SkyCoord` instance

The frame to transform this coordinate into.

Returns

coord : `SkyCoord`

A new object with this coordinate represented in the `frame` frame.

Raises**ValueError**

If there is no possible transformation route.

SphericalRepresentation

class `astropy.coordinates.SphericalRepresentation` (*lon, lat, distance, copy=True*)

Bases: `astropy.coordinates.BaseRepresentation`

Representation of points in 3D spherical coordinates.

Parameters

lon, lat : `Quantity`

The longitude and latitude of the point(s), in angular units. The latitude should be between -90 and 90 degrees, and the longitude will be wrapped to an angle between 0 and 360 degrees. These can also be instances of `Angle`, `Longitude`, or `Latitude`.

distance : `Quantity`

The distance to the point(s). If the distance is a length, it is passed to the `Distance` class, otherwise it is passed to the `Quantity` class.

copy : bool, optional

If True arrays will be copied rather than referenced.

Attributes Summary

<code>attr_classes</code>	Dictionary that remembers insertion order
<code>distance</code>	The distance from the origin to the point(s).
<code>lat</code>	The latitude of the point(s).
<code>lon</code>	The longitude of the point(s).
<code>recommended_units</code>	

Methods Summary

<code>from_cartesian(cart)</code>	Converts 3D rectangular cartesian coordinates to spherical polar coordinates.
<code>represent_as(other_class)</code>	
<code>to_cartesian()</code>	Converts spherical polar coordinates to 3D rectangular cartesian coordinates.

Attributes Documentation

attr_classes = `OrderedDict([(u'lon', <class 'astropy.coordinates.angles.Longitude'>), (u'lat', <class 'astropy.coordina`

distance

The distance from the origin to the point(s).

lat

The latitude of the point(s).

lon

The longitude of the point(s).

recommended_units = `{u'lat': Unit("deg"), u'lon': Unit("deg")}`

Methods Documentation

classmethod `from_cartesian` (*cart*)

Converts 3D rectangular cartesian coordinates to spherical polar coordinates.

represent_as (*other_class*)

to_cartesian ()

Converts spherical polar coordinates to 3D rectangular cartesian coordinates.

StaticMatrixTransform

class `astropy.coordinates.StaticMatrixTransform` (*matrix, fromsys, tosys, priority=1, register_graph=None*)

Bases: `astropy.coordinates.CoordinateTransform`

A coordinate transformation defined as a 3 x 3 cartesian transformation matrix.

This is distinct from `DynamicMatrixTransform` in that this kind of matrix is independent of frame attributes. That is, it depends *only* on the class of the frame.

Parameters

matrix : array-like or callable

A 3 x 3 matrix for transforming 3-vectors. In most cases will be unitary (although this is not strictly required). If a callable, will be called *with no arguments* to get the matrix.

fromsys : class

The coordinate frame class to start from.

tosys : class

The coordinate frame class to transform into.

priority : number

The priority if this transform when finding the shortest coordinate transform path - large numbers are lower priorities.

register_graph : `TransformGraph` or `None`

A graph to register this transformation with on creation, or `None` to leave it unregistered.

Raises

ValueError

If the matrix is not 3 x 3

Methods Summary

`__call__` (*fromcoord, toframe*)

Methods Documentation

`__call__` (*fromcoord, toframe*)

TimeFrameAttribute

class `astropy.coordinates.TimeFrameAttribute` (*default=None, secondary_attribute=u''*)

Bases: `astropy.coordinates.FrameAttribute`

Frame attribute descriptor for quantities that are Time objects. See the `FrameAttribute` API doc for further information.

Parameters

default : object

Default value for the attribute if not provided

secondary_attribute : str

Name of a secondary instance attribute which supplies the value if `default` is `None` and no value was supplied during initialization.

Returns

frame_attr : descriptor

A new data descriptor to hold a frame attribute

Methods Summary

`convert_input(value)` Convert input value to a Time object and validate by running through the Time constructor.

Methods Documentation

convert_input (*value*)

Convert input value to a Time object and validate by running through the Time constructor. Also check that the input was a scalar.

Parameters

value : object

Input value to be converted.

Returns

out, converted : correctly-typed object, boolean

Tuple consisting of the correctly-typed object and a boolean which indicates if conversion was actually performed.

Raises

ValueError

If the input is not valid for this attribute.

TransformGraph

class `astropy.coordinates.TransformGraph`

Bases: `object`

A graph representing the paths between coordinate frames.

Attributes Summary

`frame_set` A set of all the frame classes present in this `TransformGraph`.

Methods Summary

<code>add_transform(fromsys, tosys, transform)</code>	Add a new coordinate transformation to the graph.
<code>find_shortest_path(fromsys, tosys)</code>	Computes the shortest distance along the transform graph from one system to another.
<code>get_names()</code>	Returns all available transform names.
<code>get_transform(fromsys, tosys)</code>	Generates and returns the <code>CompositeTransform</code> for a transformation between two systems.
<code>invalidate_cache()</code>	Invalidates the cache that stores optimizations for traversing the transform graph.
<code>lookup_name(name)</code>	Tries to locate the coordinate class with the provided alias.
<code>remove_transform(fromsys, tosys, transform)</code>	Removes a coordinate transform from the graph.
<code>to_dot_graph([priorities, addnodes, savefn, ...])</code>	Converts this transform graph to the <code>graphviz</code> DOT format.
<code>to_networkx_graph()</code>	Converts this transform graph into a <code>networkx</code> graph.
<code>transform(transcls, fromsys, tosys[, priority])</code>	A function decorator for defining transformations.

Attributes Documentation

`frame_set`

A set of all the frame classes present in this `TransformGraph`.

Methods Documentation

`add_transform` (*fromsys, tosys, transform*)

Add a new coordinate transformation to the graph.

Parameters

fromsys : class

The coordinate frame class to start from.

tosys : class

The coordinate frame class to transform into.

transform : `CoordinateTransform` or similar callable

The transformation object. Typically a `CoordinateTransform` object, although it may be some other callable that is called with the same signature.

Raises

TypeError

If `fromsys` or `tosys` are not classes or `transform` is not callable.

`find_shortest_path` (*fromsys, tosys*)

Computes the shortest distance along the transform graph from one system to another.

Parameters

fromsys : class

The coordinate frame class to start from.

tosys : class

The coordinate frame class to transform into.

Returns

path : list of classes or `None`

The path from `fromsys` to `tosys` as an in-order sequence of classes. This list includes *both* `fromsys` and `tosys`. Is `None` if there is no possible path.

distance : number

The total distance/priority from `fromsys` to `tosys`. If priorities are not set this is the number of transforms needed. Is `inf` if there is no possible path.

get_names ()

Returns all available transform names. They will all be valid arguments to `lookup_name`.

Returns

nms : list

The aliases for coordinate systems.

get_transform (*fromsys*, *tosys*)

Generates and returns the `CompositeTransform` for a transformation between two coordinate systems.

Parameters

fromsys : class

The coordinate frame class to start from.

tosys : class

The coordinate frame class to transform into.

Returns

trans : `CompositeTransform` or `None`

If there is a path from `fromsys` to `tosys`, this is a transform object for that path. If no path could be found, this is `None`.

Notes

This function always returns a `CompositeTransform`, because `CompositeTransform` is slightly more adaptable in the way it can be called than other transform classes. Specifically, it takes care of inetermediate steps of transformations in a way that is consistent with 1-hop transformations.

invalidate_cache ()

Invalidates the cache that stores optimizations for traversing the transform graph. This is called automatically when transforms are added or removed, but will need to be called manually if weights on transforms are modified inplace.

lookup_name (*name*)

Tries to locate the coordinate class with the provided alias.

Parameters

name : str

The alias to look up.

Returns

coordcls

The coordinate class corresponding to the `name` or `None` if no such class exists.

remove_transform (*fromsys, tosys, transform*)

Removes a coordinate transform from the graph.

Parameters

fromsys : class or `None`

The coordinate frame *class* to start from. If `None`, `transform` will be searched for and removed (`tosys` must also be `None`).

tosys : class or `None`

The coordinate frame *class* to transform into. If `None`, `transform` will be searched for and removed (`fromsys` must also be `None`).

transform : callable or `None`

The transformation object to be removed or `None`. If `None` and `tosys` and `fromsys` are supplied, there will be no check to ensure the correct object is removed.

to_dot_graph (*priorities=True, addnodes=[], savefn=None, savelayout=u'plain', saveformat=None*)

Converts this transform graph to the `graphviz` DOT format.

Optionally saves it (requires `graphviz` be installed and on your path).

Parameters

priorities : bool

If `True`, show the priority values for each transform. Otherwise, they will not be included in the graph.

addnodes : sequence of str

Additional coordinate systems to add (this can include systems already in the transform graph, but they will only appear once).

savefn : `None` or str

The file name to save this graph to or `None` to not save to a file.

savelayout : str

The `graphviz` program to use to layout the graph (see `graphviz` for details) or 'plain' to just save the DOT graph content. Ignored if `savefn` is `None`.

saveformat : str

The `graphviz` output format. (e.g. the `-Txxx` option for the command line program - see `graphviz` docs for details). Ignored if `savefn` is `None`.

Returns

dotgraph : str

A string with the DOT format graph.

to_networkx_graph ()

Converts this transform graph into a `networkx` graph.

Note: You must have the `networkx` package installed for this to work.

Returns

nxgraph : `networkx.Graph`

This `TransformGraph` as a `networkx.Graph`.

transform (*transcls, fromsys, tosys, priority=1*)
 A function decorator for defining transformations.

Note: If decorating a static method of a class, `@staticmethod` should be added *above* this decorator.

Parameters

transcls : class

The class of the transformation object to create.

fromsys : class

The coordinate frame class to start from.

tosys : class

The coordinate frame class to transform into.

priority : number

The priority if this transform when finding the shortest coordinate transform path - large numbers are lower priorities.

Returns

deco : function

A function that can be called on another function as a decorator (see example).

Notes

This decorator assumes the first argument of the `transcls` initializer accepts a callable, and that the second and third are `fromsys` and `tosys`. If this is not true, you should just initialize the class manually and use `add_transform` instead of using this decorator.

Examples

```
graph = TransformGraph()

class Frame1(BaseCoordinateFrame):
    ...

class Frame2(BaseCoordinateFrame):
    ...

@graph.transform(FunctionTransform, Frame1, Frame2)
def f1_to_f2(f1_obj):
    ... do something with f1_obj ...
    return f2_obj
```

UnitSphericalRepresentation

class `astropy.coordinates.UnitSphericalRepresentation` (*lon, lat, copy=True*)

Bases: `astropy.coordinates.BaseRepresentation`

Representation of points on a unit sphere.

Parameters**lon, lat** : *Quantity* or str

The longitude and latitude of the point(s), in angular units. The latitude should be between -90 and 90 degrees, and the longitude will be wrapped to an angle between 0 and 360 degrees. These can also be instances of *Angle*, *Longitude*, or *Latitude*.

copy : bool, optional

If True arrays will be copied rather than referenced.

Attributes Summary

<code>attr_classes</code>	Dictionary that remembers insertion order
<code>lat</code>	The latitude of the point(s).
<code>lon</code>	The longitude of the point(s).
<code>recommended_units</code>	

Methods Summary

<code>from_cartesian(cart)</code>	Converts 3D rectangular cartesian coordinates to spherical polar coordinates.
<code>represent_as(other_class)</code>	
<code>to_cartesian()</code>	Converts spherical polar coordinates to 3D rectangular cartesian coordinates.

Attributes Documentation**attr_classes** = `OrderedDict([(u'lon', <class 'astropy.coordinates.angles.Longitude'>), (u'lat', <class 'astropy.coordina`)**lat**

The latitude of the point(s).

lon

The longitude of the point(s).

recommended_units = `{u'lat': Unit("deg"), u'lon': Unit("deg")}`**Methods Documentation****classmethod from_cartesian** (*cart*)

Converts 3D rectangular cartesian coordinates to spherical polar coordinates.

represent_as (*other_class*)**to_cartesian** ()

Converts spherical polar coordinates to 3D rectangular cartesian coordinates.

Class Inheritance Diagram

WORLD COORDINATE SYSTEM (`ASTROPY.WCS`)

11.1 Introduction

`astropy.wcs` contains utilities for managing World Coordinate System (WCS) transformations in FITS files. These transformations map the pixel locations in an image to their real-world units, such as their position on the sky sphere.

It is at its base a wrapper around Mark Calabretta's `wcslib`, but also adds support for the Simple Imaging Polynomial (SIP) convention and table lookup distortions as defined in WCS Paper IV. Each of these transformations can be used independently or together in a standard pipeline.

11.2 Getting Started

The basic workflow is as follows:

1. `from astropy import wcs`
2. Call the `WCS` constructor with an `astropy.io.fits` header and/or `hdulist` object.
3. Optionally, if the FITS file uses any deprecated or non-standard features, you may need to call one of the `fix` methods on the object.
4. Use one of the following transformation methods:
 - `all_pix2world`: Perform all three transformations from pixel to world coordinates.
 - `wcs_pix2world`: Perform just the core WCS transformation from pixel to world coordinates.
 - `all_world2pix`: Perform all three transformations from world to pixel coordinates, using an iterative method if necessary.
 - `wcs_world2pix`: Perform just the core WCS transformation from world to pixel coordinates.
 - `sip_pix2foc`: Convert from pixel to focal plane coordinates using the SIP polynomial coefficients.
 - `sip_foc2pix`: Convert from focal plane to pixel coordinates using the SIP polynomial coefficients.
 - `p4_pix2foc`: Convert from pixel to focal plane coordinates using the table lookup distortion method described in Paper IV.
 - `det2im`: Convert from detector coordinates to image coordinates. Commonly used for narrow column correction.

For example, to convert pixel coordinates to world coordinates:

```
>>> from astropy.wcs import WCS
>>> w = WCS('image.fits')
>>> lon, lat = w.all_pix2world(30, 40, 0)
>>> print(lon, lat)
```

11.3 Using `astropy.wcs`

11.3.1 Loading WCS information from a FITS file

This example loads a FITS file (supplied on the commandline) and uses the WCS cards in its primary header to transform.

```
# Load the WCS information from a fits header, and use it
# to convert pixel coordinates to world coordinates.

from __future__ import division, print_function

import numpy
from astropy import wcs
from astropy.io import fits
import sys

def load_wcs_from_file(filename):
    # Load the FITS hdulist using astropy.io.fits
    hdulist = fits.open(sys.argv[-1])

    # Parse the WCS keywords in the primary HDU
    w = wcs.WCS(hdulist[0].header)

    # Print out the "name" of the WCS, as defined in the FITS header
    print(w.wcs.name)

    # Print out all of the settings that were parsed from the header
    w.wcs.print_contents()

    # Some pixel coordinates of interest.
    pixcrd = numpy.array([[0, 0], [24, 38], [45, 98]], numpy.float_)

    # Convert pixel coordinates to world coordinates
# The second argument is "origin" -- in this case we're declaring we
# have 1-based (Fortran-like) coordinates.
    world = w.wcs_pix2world(pixcrd, 1)
    print(world)

    # Convert the same coordinates back to pixel coordinates.
    pixcrd2 = w.wcs_world2pix(world, 1)
    print(pixcrd2)

    # These should be the same as the original pixel coordinates, modulo
# some floating-point error.
    assert numpy.max(numpy.abs(pixcrd - pixcrd2)) < 1e-6

if __name__ == '__main__':
    load_wcs_from_file(sys.argv[-1])
```

11.3.2 Building a WCS structure programmatically

This example, rather than starting from a FITS header, sets WCS values programmatically, uses those settings to transform some points, and then saves those settings to a new FITS header.

```
# Set the WCS information manually by setting properties of the WCS
# object.

from __future__ import division, print_function

import numpy
from astropy import wcs
from astropy.io import fits

# Create a new WCS object. The number of axes must be set
# from the start
w = wcs.WCS(naxis=2)

# Set up an "Airy's zenithal" projection
# Vector properties may be set with Python lists, or Numpy arrays
w.wcs.crpix = [-234.75, 8.3393]
w.wcs.cdelt = numpy.array([-0.0666667, 0.0666667])
w.wcs.crval = [0, -90]
w.wcs.ctype = ["RA---AIR", "DEC--AIR"]
w.wcs.set_pv([(2, 1, 45.0)])

# Some pixel coordinates of interest.
pixcrd = numpy.array([[0, 0], [24, 38], [45, 98]], numpy.float_)

# Convert pixel coordinates to world coordinates
world = w.wcs_pix2world(pixcrd, 1)
print(world)

# Convert the same coordinates back to pixel coordinates.
pixcrd2 = w.wcs_world2pix(world, 1)
print(pixcrd2)

# These should be the same as the original pixel coordinates, modulo
# some floating-point error.
assert numpy.max(numpy.abs(pixcrd - pixcrd2)) < 1e-6

# Now, write out the WCS object as a FITS header
header = w.to_header()

# header is an astropy.io.fits.Header object. We can use it to create a new
# PrimaryHDU and write it to a file.
hdu = fits.PrimaryHDU(header=header)
# Save to FITS file
# hdu.writeto('test.fits')
```

11.3.3 Validating the WCS keywords in a FITS file

Astropy includes a commandline tool, `wcslint` to check the WCS keywords in a FITS file:

```
> wcslint invalid.fits
HDU 1:
  WCS key ' ':
```

- RADECSYS= 'ICRS ' / Astrometric system
RADECSYS is non-standard, use RADESYSa.
- The WCS transformation has more axes (2) than the image it is associated with (0)
- 'celfix' made the change 'PV1_5 : Unrecognized coordinate transformation parameter'.

HDU 2:

- WCS key ' ':
 - The WCS transformation has more axes (3) than the image it is associated with (0)
 - 'celfix' made the change 'In CUNIT2 : Mismatched units type 'length': have 'Hz', want 'm''.
 - 'unitfix' made the change 'Changed units: 'HZ ' -> 'Hz''.

11.3.4 Bounds checking

Bounds checking is enabled by default, and any computed world coordinates outside of $[-180^\circ, 180^\circ]$ for longitude and $[-90^\circ, 90^\circ]$ in latitude are marked as invalid. To disable this behavior, use `astropy.wcs.Wcsprm.bounds_check`.

11.4 Supported projections

As `astropy.wcs` is based on `wcslib`, it supports the standard projections defined in the WCS papers. These projection codes are specified in the second part of the `CUNITn` keywords (accessible through `Wcsprm.cunit`), for example, `RA-TAN-SIP`. The supported projection codes are:

- AZP: zenithal/azimuthal perspective
- SZP: slant zenithal perspective
- TAN: gnomonic
- STG: stereographic
- SIN: orthographic/synthesis
- ARC: zenithal/azimuthal equidistant
- ZPN: zenithal/azimuthal polynomial
- ZEA: zenithal/azimuthal equal area
- AIR: Airy's projection
- CYP: cylindrical perspective
- CEA: cylindrical equal area
- CAR: plate carrée
- MER: Mercator's projection
- COP: conic perspective
- COE: conic equal area
- COD: conic equidistant
- COO: conic orthomorphic

- SFL: Sanson-Flamsteed (“global sinusoid”)
- PAR: parabolic
- MOL: Mollweide’s projection
- AIT: Hammer-Aitoff
- BON: Bonne’s projection
- PCO: polyconic
- TSC: tangential spherical cube
- CSC: COBE quadrilateralized spherical cube
- QSC: quadrilateralized spherical cube
- HPX: HEALPix
- XPH: HEALPix polar, aka “butterfly”

11.5 Other information

11.5.1 Relax constants

The `relax` keyword argument controls the handling of non-standard FITS WCS keywords.

Note that the default value of `relax` is `True` for reading (to accept all non standard keywords), and `False` for writing (to write out only standard keywords), in accordance with [Postel’s prescription](#):

“Be liberal in what you accept, and conservative in what you send.”

Header-reading relaxation constants

`WCS`, `Wcsprm` and `find_all_wcs` have a `relax` argument, which may be either `True`, `False` or an `int`.

- If `True`, (default), all non-standard WCS extensions recognized by the parser will be handled.
- If `False`, none of the extensions (even those in the errata) will be handled. Non-conformant keywords will be handled in the same way as non-WCS keywords in the header, i.e. by simply ignoring them.
- If an `int`, is a bit field to provide fine-grained control over what non-standard WCS keywords to accept. The flag bits are subject to change in future and should be set by using the constants beginning with `WCSHDR_` in the `astropy.wcs` module.

For example, to accept `CD00i00j` and `PC00i00j` use:

```
relax = astropy.wcs.WCSHDR_CD00i00j | astropy.wcs.WCSHDR_PC00i00j
```

The parser always treats `EPOCH` as subordinate to `EQUINOXa` if both are present, and `VSOURCEa` is always subordinate to `ZSOURCEa`.

Likewise, `VELREF` is subordinate to the formalism of WCS Paper III.

The flag bits are:

- `WCSHDR_none`: Don’t accept any extensions (not even those in the errata). Treat non-conformant keywords in the same way as non-WCS keywords in the header, i.e. simply ignore them. (This is equivalent to passing `False`)

- `WCSHDR_all`: Accept all extensions recognized by the parser. (This is equivalent to the default behavior or passing `True`).
- `WCSHDR_CROTAia`: Accept `CROTAia`, `icROTna`, `TCROTna`
- `WCSHDR_EPOCHa`: Accept `EPOCHa`.
- `WCSHDR_VELREFa`: Accept `VELREFa`.

The constructor always recognizes the AIPS-convention keywords, `CROTAn`, `EPOCH`, and `VELREF` for the primary representation (`a = ' '`) but alternates are non-standard.

The constructor accepts `EPOCHa` and `VELREFa` only if `WCSHDR_AUXIMG` is also enabled.

- `WCSHDR_CD00i00j`: Accept `CD00i00j`.
- `WCSHDR_PC00i00j`: Accept `PC00i00j`.
- `WCSHDR_PROJpn`: Accept `PROJpn`.

These appeared in early drafts of WCS Paper I+II (before they were split) and are equivalent to `CDi_ja`, `PCi_ja`, and `PVi_ma` for the primary representation (`a = ' '`). `PROJpn` is equivalent to `PVi_ma` with $m = n \leq 9$, and is associated exclusively with the latitude axis.

- `WCSHDR_RADECSYS`: Accept `RADECSYS`. This appeared in early drafts of WCS Paper I+II and was subsequently replaced by `RADESYSa`. The constructor accepts `RADECSYS` only if `WCSHDR_AUXIMG` is also enabled.
- `WCSHDR_VSOURCE`: Accept `VSOURCEa` or `VSOUna`. This appeared in early drafts of WCS Paper III and was subsequently dropped in favour of `ZSOURCEa` and `ZSOUna`. The constructor accepts `VSOURCEa` only if `WCSHDR_AUXIMG` is also enabled.
- `WCSHDR_DOBSn`: Allow `DOBSn`, the column-specific analogue of `DATE-OBS`. By an oversight this was never formally defined in the standard.
- `WCSHDR_LONGKEY`: Accept long forms of the alternate binary table and pixel list WCS keywords, i.e. with “a” non-blank. Specifically:

```

jCRPXna  TCRPXna  :  jCRPXn  jCRPna  TCRPXn  TCRPna  CRPIXja
-         TPCn_ka  :  -         ijPCna  -         TPn_ka  PCi_ja
-         TCDn_ka  :  -         ijCDna  -         TCn_ka  CDi_ja
iCDLTna  TCDLTna  :  iCDLTn  iCDEna  TCDLTn  TCDEna  CDELTia
iCUNIna  TCUNIna  :  iCUNIn  iCUNna  TCUNIn  TCUNna  CUNITia
iCTYPna  TCTYPna  :  iCTYPn  iCTYna  TCTYPn  TCTYna  CTYPeia
iCRVLna  TCRVLna  :  iCRVLn  iCRVna  TCRVLn  TCRVna  CRVALia
iPVn_ma  TPVn_ma  :  -         iVn_ma  -         TVn_ma  PVi_ma
iPSn_ma  TPSn_ma  :  -         iSn_ma  -         TSn_ma  PSi_ma

```

where the primary and standard alternate forms together with the image-header equivalent are shown rightwards of the colon.

The long form of these keywords could be described as quasi-standard. `TPCn_ka`, `iPVn_ma`, and `TPVn_ma` appeared by mistake in the examples in WCS Paper II and subsequently these and also `TCDn_ka`, `iPSn_ma` and `TPSn_ma` were legitimized by the errata to the WCS papers.

Strictly speaking, the other long forms are non-standard and in fact have never appeared in any draft of the WCS papers nor in the errata. However, as natural extensions of the primary form they are unlikely to be written with any other intention. Thus it should be safe to accept them provided, of course, that the resulting keyword does not exceed the 8-character limit.

If `WCSHDR_CNAMn` is enabled then also accept:

```

iCNAMna  TCNAMna  :  ---  iCNAna  ---  TCNAAna  CNAMEia
iCRDEna  TCRDEna  :  ---  iCRDna  ---  TCRDna  CRDERia
iCSYEna  TCSYEna  :  ---  iCSYna  ---  TCSYna  CSYERia

```

Note that CNAME_{ia}, CRDER_{ia}, CSYER_{ia}, and their variants are not used by `astropy.wcs` but are stored as auxiliary information.

- WCSHDR_CNAM_n: Accept `iCNAMn`, `iCRDEn`, `iCSYEn`, `TCNAMn`, `TCRDEn`, and `TCSYEn`, i.e. with a blank. While non-standard, these are the obvious analogues of `iCTYPn`, `TCTYPn`, etc.
- WCSHDR_AUXIMG: Allow the image-header form of an auxiliary WCS keyword with representation-wide scope to provide a default value for all images. This default may be overridden by the column-specific form of the keyword.

For example, a keyword like EQUINOX_a would apply to all image arrays in a binary table, or all pixel list columns with alternate representation `a` unless overridden by EQUIN_a.

Specifically the keywords are:

```
LATPOLEa  for LATPna
LONPOLEa  for LONPna
RESTFREQ  for RFRQna
RESTFRQa  for RFRQna
RESTWAVa  for RWAVna
```

whose keyvalues are actually used by WCSLIB, and also keywords that provide auxiliary information that is simply stored in the `wcsprm` struct:

```
EPOCH      -          ... (No column-specific form.)
EPOCHa    -          ... Only if WSHDR_EPOCHa is set.
EQUINOXa  for EQUINa
RADESYSa  for RADEna
RADECSYS   for RADEna ... Only if WSHDR_RADECSYS is set.
SPECSYSa  for SPECna
SSYSOBSa  for SOBSna
SSYSSRCa  for SSRCna
VELOSYSa  for VYSna
VELANGLa  for VANGLna
VELREF     -          ... (No column-specific form.)
VELREFa   -          ... Only if WSHDR_VELREFa is set.
VSOURCEa  for VSOUna ... Only if WSHDR_VSOURCE is set.
WCSNAMEa  for WCSNna ... Or TWCSna (see below).
ZSOURCEa  for ZSOUna

DATE-AVG   for DAVGn
DATE-OBS   for DOBSn
MJD-AVG    for MJDAN
MJD-OBS    for MJDON
OBSGEO-X   for OBSGXn
OBSGEO-Y   for OBSGYn
OBSGEO-Z   for OBSGZn
```

where the image-header keywords on the left provide default values for the column specific keywords on the right.

Keywords in the last group, such as MJD-OBS, apply to all alternate representations, so MJD-OBS would provide a default value for all images in the header.

This auxiliary inheritance mechanism applies to binary table image arrays and pixel lists alike. Most of these keywords have no default value, the exceptions being LONPOLE_a and LATPOLE_a, and also RADESYS_a and EQUINOX_a which provide defaults for each other. Thus the only potential difficulty in using WSHDR_AUXIMG is that of erroneously inheriting one of these four keywords.

Unlike WSHDR_ALLIMG, the existence of one (or all) of these auxiliary WCS image header keywords will not by itself cause a `Wcsprm` object to be created for alternate representation `a`. This is because they do not provide

sufficient information to create a non-trivial coordinate representation when used in conjunction with the default values of those keywords, such as `CTYPEia`, that are parameterized by axis number.

- `WCSHDR_ALLIMG`: Allow the image-header form of *all* image header WCS keywords to provide a default value for all image arrays in a binary table (n.b. not pixel list). This default may be overridden by the column-specific form of the keyword.

For example, a keyword like `CRPIXja` would apply to all image arrays in a binary table with alternate representation `a` unless overridden by `jCRPna`.

Specifically the keywords are those listed above for `WCSHDR_AUXIMG` plus:

```
WCSAXESa  for WCAXna
```

which defines the coordinate dimensionality, and the following keywords which are parameterized by axis number:

```
CRPIXja   for jCRPna
PCi_ja    for ijPCna
CDi_ja    for ijCDna
CDELTia   for iCDEna
CROTAi    for iCROtn
CROTAia   -           ... Only if WCSHDR_CROTAia is set.
CUNITia   for iCUNna
CTYPEia   for iCTYna
CRVALia   for iCRVna
PVi_ma    for iVn_ma
PSi_ma    for iSn_ma

CNAMEia   for iCNAna
CRDERia   for iCRDna
CSYERia   for iCSYna
```

where the image-header keywords on the left provide default values for the column specific keywords on the right.

This full inheritance mechanism only applies to binary table image arrays, not pixel lists, because in the latter case there is no well-defined association between coordinate axis number and column number.

Note that `CNAMEia`, `CRDERia`, `CSYERia`, and their variants are not used by `pywcs` but are stored in the `Wcsprm` object as auxiliary information.

Note especially that at least one `Wcsprm` object will be returned for each `a` found in one of the image header keywords listed above:

- If the image header keywords for a **are not** inherited by a binary table, then the struct will not be associated with any particular table column number and it is up to the user to provide an association.
- If the image header keywords for a **are** inherited by a binary table image array, then those keywords are considered to be “exhausted” and do not result in a separate `Wcsprm` object.

Header-writing relaxation constants

`to_header` and `to_header_string` has a *relax* argument which may be either `True`, `False` or an `int`.

- If `True`, write all recognized extensions.
- If `False` (default), write all extensions that are considered to be safe and recommended, equivalent to `WCSHDO_safe` (described below).

- If an `int`, is a bit field to provide fine-grained control over what non-standard WCS keywords to accept. The flag bits are subject to change in future and should be set by using the constants beginning with `WCSHDO_` in the `astropy.wcs` module.

The flag bits are:

- `WCSHDO_none`: Don't use any extensions.
- `WCSHDO_all`: Write all recognized extensions, equivalent to setting each flag bit.
- `WCSHDO_safe`: Write all extensions that are considered to be safe and recommended.
- `WCSHDO_DOBSn`: Write `DOBSn`, the column-specific analogue of `DATE-OBS` for use in binary tables and pixel lists. `WCS Paper III` introduced `DATE-AVG` and `DAVGn` but by an oversight `DOBSn` (the obvious analogy) was never formally defined by the standard. The alternative to using `DOBSn` is to write `DATE-OBS` which applies to the whole table. This usage is considered to be safe and is recommended.
- `WCSHDO_TPCn_ka`: `WCS Paper I` defined
 - `TPn_ka` and `TCn_ka` for pixel lists
but `WCS Paper II` uses `TPCn_ka` in one example and subsequently the errata for the `WCS` papers legitimized the use of
 - `TPCn_ka` and `TCDn_ka` for pixel lists
provided that the keyword does not exceed eight characters. This usage is considered to be safe and is recommended because of the non-mnemonic terseness of the shorter forms.
- `WCSHDO_PVn_ma`: `WCS Paper I` defined
 - `iVn_ma` and `iSn_ma` for bintables and
 - `TVn_ma` and `TSn_ma` for pixel lists
but `WCS Paper II` uses `iPVn_ma` and `TPVn_ma` in the examples and subsequently the errata for the `WCS` papers legitimized the use of
 - `iPVn_ma` and `iPSn_ma` for bintables and
 - `TPVn_ma` and `TPSn_ma` for pixel lists
provided that the keyword does not exceed eight characters. This usage is considered to be safe and is recommended because of the non-mnemonic terseness of the shorter forms.
- `WCSHDO_CRPXna`: For historical reasons `WCS Paper I` defined
 - `jCRPXn`, `iCDLTn`, `iCUNIn`, `iCTYPn`, and `iCRVLn` for bintables and
 - `TCRPXn`, `TCDLTn`, `TCUNIn`, `TCTYPn`, and `TCRVLn` for pixel lists
for use without an alternate version specifier. However, because of the eight-character keyword constraint, in order to accommodate column numbers greater than 99 `WCS Paper I` also defined
 - `jCRPna`, `iCDEna`, `iCUNna`, `iCTYna` and `iCRVna` for bintables and
 - `TCRPna`, `TCDEna`, `TCUNna`, `TCTYna` and `TCRVna` for pixel lists
for use with an alternate version specifier (the `a`). Like the `PC`, `CD`, `PV`, and `PS` keywords there is an obvious tendency to confuse these two forms for column numbers up to 99. It is very unlikely that any parser would reject keywords in the first set with a non-blank alternate version specifier so this usage is considered to be safe and is recommended.
- `WCSHDO_CNAMna`: `WCS Papers I` and `III` defined
 - `iCNAna`, `iCRDna`, and `iCSYna` for bintables and

- TCNAna, TCRDna, and TCSYna for pixel lists

By analogy with the above, the long forms would be

- iCNAMna, iCRDEna, and iCSYEna for bintables and
- TCNAMna, TCRDEna, and TCSYEna for pixel lists

Note that these keywords provide auxiliary information only, none of them are needed to compute world coordinates. This usage is potentially unsafe and is not recommended at this time.

- WCSHDO_WCSNna: Write WCSNna instead of TWCSna for pixel lists. While the constructor treats WCSNna and TWCSna as equivalent, other parsers may not. Consequently, this usage is potentially unsafe and is not recommended at this time.
- WCSHDO_SIP: Write out Simple Imaging Polynomial (SIP) keywords.

11.5.2 astropy.wcs History

`astropy.wcs` began life as `pywcs`. Earlier version numbers refer to that package.

pywcs Version 1.11

- Updated to `wcslib` version 4.8, which gives much more detailed error messages.
- Added functions `get_pc()` and `get_cdelt()`. These provide a way to always get the canonical representation of the linear transformation matrix, whether the header specified it in PC, CD or CROTA form.
- Long-running process will now release the Python GIL to better support Python multithreading.
- The dimensions of the `cd` and `pc` matrices were always returned as 2x2. They now are sized according to `naxis`.
- Supports Python 3.x
- Builds on Microsoft Windows without severely patching `wcslib`.
- Lots of new unit tests
- `pywcs` will now run without `pyfits`, though the SIP and distortion lookup table functionality is unavailable.
- Setting `cunit` will now verify that the values are valid unit strings.

pywcs Version 1.10

- Adds a `UnitConversion` class, which gives access to `wcslib`'s unit conversion functionality. Given two convertible unit strings, `pywcs` can convert arrays of values from one to the other.
- Now uses `wcslib` 4.7
- Changes to some `wcs` values would not always calculate secondary values.

pywcs Version 1.9

- Support binary image arrays and pixel list format WCS by presenting a way to call `wcslib`'s `wcsbth()`
- Updated underlying `wcslib` to version 4.5, which fixes the following:

- Fixed the interpretation of VELREF when translating AIPS-convention spectral types. Such translation is now handled by a new special- purpose function, `scaips()`. The `wcsprm` struct has been augmented with an entry for `velref` which is filled by `wcspih()` and `wcsbth()`. Previously, selection by VELREF of the radio or optical velocity convention for type VELO was not properly handled.

Bugs

- The `pc` member is now available with a default raw `Wcsprm` object.
- Make properties that return arrays read-only, since modifying a (mutable) array could result in secondary values not being recomputed based on those changes.
- `float` properties can now be set using `int` values

pywcs Version 1.3a1

Earlier versions of `pywcs` had two versions of every conversion method:

```
X(...)      -- treats the origin of pixel coordinates at (0, 0)
X_fits(...) -- treats the origin of pixel coordinates at (1, 1)
```

From version 1.3 onwards, there is only one method for each conversion, with an ‘origin’ argument:

- 0: places the origin at (0, 0), which is the C/Numpy convention.
- 1: places the origin at (1, 1), which is the Fortran/FITS convention.

11.6 See Also

- `wcslib`

11.7 Reference/API

11.7.1 `astropy.wcs` Module

Introduction

`astropy.wcs` contains utilities for managing World Coordinate System (WCS) transformations in FITS files. These transformations map the pixel locations in an image to their real-world units, such as their position on the sky sphere.

It is at its base a wrapper around Mark Calabretta’s `wcslib`, but also adds support for the Simple Imaging Polynomial (SIP) convention and table lookup distortions as defined in WCS Paper IV. Each of these transformations can be used independently or together in a standard pipeline.

Functions

<code>find_all_wcs(header[, relax, keyset, fix, ...])</code>	Find all the WCS transformations in the given header.
<code>get_include()</code>	Get the path to <code>astropy.wcs</code> ’s C header files.
<code>validate(source)</code>	Prints a WCS validation report for the given FITS file.

find_all_wcs

```
astropy.wcs.find_all_wcs(header, relax=True, keyset=None, fix=True, translate_units='u',
                        _do_set=True)
```

Find all the WCS transformations in the given header.

Parameters

header : str or astropy.io.fits header object.

relax : bool or int, optional

Degree of permissiveness:

- `True` (default): Admit all recognized informal extensions of the WCS standard.
- `False`: Recognize only FITS keywords defined by the published WCS standard.
- `int`: a bit field selecting specific extensions to accept. See *Header-reading relaxation constants* for details.

keyset : sequence of flags, optional

A list of flags used to select the keyword types considered by `wcslib`. When `None`, only the standard image header keywords are considered (and the underlying `wcsphih()` C function is called). To use binary table image array or pixel list keywords, `keyset` must be set.

Each element in the list should be one of the following strings:

- `'image'`: Image header keywords
- `'binary'`: Binary table image array keywords
- `'pixel'`: Pixel list keywords

Keywords such as `EQUIna` or `RFRQna` that are common to binary table image arrays and pixel lists (including `WCSNna` and `TWCSna`) are selected by both `'binary'` and `'pixel'`.

fix : bool, optional

When `True` (default), call `fix` on the resulting objects to fix any non-standard uses in the header. `FITSFixedWarning` warnings will be emitted if any changes were made.

translate_units : str, optional

Specify which potentially unsafe translations of non-standard unit strings to perform. By default, performs none. See `WCS.fix` for more information about this parameter. Only effective when `fix` is `True`.

Returns

wcses : list of `WCS` objects

get_include

```
astropy.wcs.get_include()
```

Get the path to `astropy.wcs`'s C header files.

validate

`astropy.wcs.validate` (*source*)

Prints a WCS validation report for the given FITS file.

Parameters

source : str path, readable file-like object or `astropy.io.fits.HDUList` object

The FITS file to validate.

Returns

results : `WcsValidateResults` instance

The result is returned as nested lists. The first level corresponds to the HDUs in the given file. The next level has an entry for each WCS found in that header. The special subclass of list will pretty-print the results as a table when printed.

Classes

<code>DistortionLookupTable</code>	Represents a single lookup table for a Paper IV distortion transformation.
<code>FITSFixedWarning</code>	The warning raised when the contents of the FITS header have been modified.
<code>InconsistentAxisTypesError</code>	The WCS header inconsistent or unrecognized coordinate axis type(s).
<code>InvalidCoordinateError</code>	One or more of the world coordinates is invalid.
<code>InvalidSubimageSpecificationError</code>	The subimage specification is invalid.
<code>InvalidTabularParametersError</code>	The given tabular parameters are invalid.
<code>InvalidTransformError</code>	The WCS transformation is invalid, or the transformation parameters are invalid.
<code>NoSolutionError</code>	No solution can be found in the given interval.
<code>NoWcsKeywordsFoundError</code>	No WCS keywords were found in the given header.
<code>NonseparableSubimageCoordinateSystemError</code>	Non-separable subimage coordinate system.
<code>SingularMatrixError</code>	The linear transformation matrix is singular.
<code>Sip</code>	The <code>Sip</code> class performs polynomial distortion correction using the SIP convention.
<code>Tabprm</code>	A class to store the information related to tabular coordinates, i.e., coordinates.
<code>WCS([header, fobj, key, minerr, relax, ...])</code>	WCS objects perform standard WCS transformations, and correct for distortions.
<code>WCSBase</code>	Wcs objects amalgamate basic WCS (as provided by wcslib), with SIP .
<code>WcsError</code>	Base class of all invalid WCS errors.
<code>Wcsprm</code>	<code>Wcsprm</code> is a direct wrapper around wcslib . It

DistortionLookupTable

class `astropy.wcs.DistortionLookupTable`

Bases: `object`

Represents a single lookup table for a [Paper IV](#) distortion transformation.

Parameters

table : 2-dimensional array

The distortion lookup table.

crpix : 2-tuple

The distortion array reference pixel

crval : 2-tuple

The image array pixel coordinate

cdelt : 2-tuple

The grid step size

Attributes Summary

<code>cdelt</code>	double array[naxis]	Coordinate increments (CDELTia) for each
<code>crpix</code>	double array[naxis]	Coordinate reference pixels (CRPIXja) for
<code>crval</code>	double array[naxis]	Coordinate reference values (CRVALia) for
<code>data</code>	float array	The array data for the

Methods Summary

<code>get_offset(x, y) -> (x, y)</code>	Returns the offset as defined in the distortion lookup table.
--	---

Attributes Documentation

cdelt

double array[naxis] Coordinate increments (CDELTia) for each coord axis.

If a CDi_ja linear transformation matrix is present, a warning is raised and `cdelt` is ignored. The CDi_ja matrix may be deleted by:

```
del wcs.wcs.cd
```

An undefined value is represented by NaN.

crpix

double array[naxis] Coordinate reference pixels (CRPIXja) for each pixel axis.

crval

double array[naxis] Coordinate reference values (CRVALia) for each coordinate axis.

data

float array The array data for the `DistortionLookupTable`.

Methods Documentation

get_offset (x, y) -> (x, y)

Returns the offset as defined in the distortion lookup table.

Returns

coordinate : coordinate pair

The offset from the distortion table for pixel point (x, y).

FITSFixedWarning

exception `astropy.wcs.FITSFixedWarning`

The warning raised when the contents of the FITS header have been modified to be standards compliant.

InconsistentAxisTypesError**exception** `astropy.wcs.InconsistentAxisTypesError`

The WCS header inconsistent or unrecognized coordinate axis type(s).

InvalidCoordinateError**exception** `astropy.wcs.InvalidCoordinateError`

One or more of the world coordinates is invalid.

InvalidSubimageSpecificationError**exception** `astropy.wcs.InvalidSubimageSpecificationError`

The subimage specification is invalid.

InvalidTabularParametersError**exception** `astropy.wcs.InvalidTabularParametersError`

The given tabular parameters are invalid.

InvalidTransformError**exception** `astropy.wcs.InvalidTransformError`

The WCS transformation is invalid, or the transformation parameters are invalid.

NoSolutionError**exception** `astropy.wcs.NoSolutionError`

No solution can be found in the given interval.

NoWcsKeywordsFoundError**exception** `astropy.wcs.NoWcsKeywordsFoundError`

No WCS keywords were found in the given header.

NonseparableSubimageCoordinateSystemError**exception** `astropy.wcs.NonseparableSubimageCoordinateSystemError`

Non-separable subimage coordinate system.

SingularMatrixError**exception** `astropy.wcs.SingularMatrixError`

The linear transformation matrix is singular.

Sip

class `astropy.wcs.Sip`

Bases: `object`

The `Sip` class performs polynomial distortion correction using the [SIP](#) convention in both directions.

Parameters

a : double array[m+1][m+1]

The A_{i_j} polynomial for pixel to focal plane transformation. Its size must be $(m + 1, m + 1)$ where $m = A_ORDER$.

b : double array[m+1][m+1]

The B_{i_j} polynomial for pixel to focal plane transformation. Its size must be $(m + 1, m + 1)$ where $m = B_ORDER$.

ap : double array[m+1][m+1]

The AP_{i_j} polynomial for pixel to focal plane transformation. Its size must be $(m + 1, m + 1)$ where $m = AP_ORDER$.

bp : double array[m+1][m+1]

The BP_{i_j} polynomial for pixel to focal plane transformation. Its size must be $(m + 1, m + 1)$ where $m = BP_ORDER$.

crpix : double array[2]

The reference pixel.

Notes

Shupe, D. L., M. Moshir, J. Li, D. Makovoz and R. Narron. 2005. "The SIP Convention for Representing Distortion in FITS Image Headers." ADASS XIV.

Attributes Summary

<code>a</code>	double array[a_order+1][a_order+1]	Focal plane transformation
<code>a_order</code>	int (read-only)	Order of the polynomial (A_ORDER).
<code>ap</code>	double array[ap_order+1][ap_order+1]	Focal plane to pixel
<code>ap_order</code>	int (read-only)	Order of the polynomial (AP_ORDER).
<code>b</code>	double array[b_order+1][b_order+1]	Pixel to focal plane
<code>b_order</code>	int (read-only)	Order of the polynomial (B_ORDER).
<code>bp</code>	double array[bp_order+1][bp_order+1]	Focal plane to pixel
<code>bp_order</code>	int (read-only)	Order of the polynomial (BP_ORDER).
<code>crpix</code>	double array[naxis]	Coordinate reference pixels (CRPIXja) for

Methods Summary

<code>foc2pix</code>	<code>sip_foc2pix(foccrd, origin) -> double array[ncoord][nelem]</code>
<code>pix2foc</code>	<code>sip_pix2foc(pixcrd, origin) -> double array[ncoord][nelem]</code>

Attributes Documentation

a

double array[a_order+1][a_order+1] Focal plane transformation matrix.

The SIP A_{i_j} matrix used for pixel to focal plane transformation.

Its values may be changed in place, but it may not be resized, without creating a new `Sip` object.

a_order

int (read-only) Order of the polynomial (A_ORDER).

ap

double array[ap_order+1][ap_order+1] Focal plane to pixel transformation matrix.

The SIP AP_{i_j} matrix used for focal plane to pixel transformation. Its values may be changed in place, but it may not be resized, without creating a new `Sip` object.

ap_order

int (read-only) Order of the polynomial (AP_ORDER).

b

double array[b_order+1][b_order+1] Pixel to focal plane transformation matrix.

The SIP B_{i_j} matrix used for pixel to focal plane transformation. Its values may be changed in place, but it may not be resized, without creating a new `Sip` object.

b_order

int (read-only) Order of the polynomial (B_ORDER).

bp

double array[bp_order+1][bp_order+1] Focal plane to pixel transformation matrix.

The SIP BP_{i_j} matrix used for focal plane to pixel transformation. Its values may be changed in place, but it may not be resized, without creating a new `Sip` object.

bp_order

int (read-only) Order of the polynomial (BP_ORDER).

crpix

double array[naxis] Coordinate reference pixels (CRPIX_ja) for each pixel axis.

Methods Documentation

foc2pix()

sip_foc2pix(*foccrd*, *origin*) -> double array[ncoord][nelem]

Convert focal plane coordinates to pixel coordinates using the SIP polynomial distortion convention.

Parameters

foccrd : double array[ncoord][nelem]

Array of focal plane coordinates.

origin : int

Specifies the origin of pixel values. The Fortran and FITS standards use an origin of 1. Numpy and C use array indexing with origin at 0.

Returns

pixcrd : double array[ncoord][nelem]

Returns an array of pixel coordinates.

Raises**MemoryError**

Memory allocation failed.

ValueError

Invalid coordinate transformation parameters.

pix2foc()

`sip_pix2foc(pixcrd, origin) -> double array[ncoord][nelem]`

Convert pixel coordinates to focal plane coordinates using the [SIP](#) polynomial distortion convention.

Parameters

pixcrd : double array[ncoord][nelem]

Array of pixel coordinates.

origin : int

Specifies the origin of pixel values. The Fortran and FITS standards use an origin of 1. Numpy and C use array indexing with origin at 0.

Returns

foccrd : double array[ncoord][nelem]

Returns an array of focal plane coordinates.

Raises**MemoryError**

Memory allocation failed.

ValueError

Invalid coordinate transformation parameters.

Tabprm

class `astropy.wcs.Tabprm`

Bases: `object`

A class to store the information related to tabular coordinates, i.e., coordinates that are defined via a lookup table.

This class can not be constructed directly from Python, but instead is returned from `tab`.

Attributes Summary

<code>K</code>	<code>int array[M]</code> (read-only) The lengths of the axes of the coordinate
<code>M</code>	<code>int</code> (read-only) Number of tabular coordinate axes.
<code>coord</code>	<code>double array[K_M] ... [K_2] [K_1] [M]</code> The tabular coordinate array.
<code>crval</code>	<code>double array[M]</code> Index values for the reference pixel for each of
<code>delta</code>	<code>double array[M]</code> (read-only) Interpolated indices into the coord
<code>extrema</code>	<code>double array[K_M] ... [K_2] [2] [M]</code> (read-only)
<code>map</code>	<code>int array[M]</code> Association between axes.
<code>nc</code>	<code>int</code> (read-only) Total number of coord vectors in the coord array.
<code>p0</code>	<code>int array[M]</code> Interpolated indices into the coordinate array.

Continued on next page

Table 11.7 – continued from previous page

<code>sense</code>	<code>int array[M]</code> +1 if monotonically increasing, -1 if decreasing.
--------------------	---

Methods Summary

<code>print_contents()</code>	Print the contents of the <code>Tabprm</code> object to stdout.
<code>set()</code>	Allocates memory for work arrays.

Attributes Documentation**K**

`int array[M]` (read-only) The lengths of the axes of the coordinate array.

An array of length `M` whose elements record the lengths of the axes of the coordinate array and of each indexing vector.

M

`int` (read-only) Number of tabular coordinate axes.

coord

`double array[K_M] ... [K_2] [K_1] [M]` The tabular coordinate array.

Has the dimensions:

`(K_M, ... K_2, K_1, M)`

(see `K`) i.e. with the `M` dimension varying fastest so that the `M` elements of a coordinate vector are stored contiguously in memory.

crval

`double array[M]` Index values for the reference pixel for each of the tabular coord axes.

delta

`double array[M]` (read-only) Interpolated indices into the coord array.

Array of interpolated indices into the coordinate array such that `Upsilon_m`, as defined in Paper III, is equal to `(p0[m] + 1) + delta[m]`.

extrema

`double array[K_M] ... [K_2] [2] [M]` (read-only)

An array recording the minimum and maximum value of each element of the coordinate vector in each row of the coordinate array, with the dimensions:

`(K_M, ... K_2, 2, M)`

(see `K`). The minimum is recorded in the first element of the compressed `K_1` dimension, then the maximum. This array is used by the inverse table lookup function to speed up table searches.

map

`int array[M]` Association between axes.

A vector of length `M` that defines the association between axis `m` in the `M`-dimensional coordinate array ($1 \leq m \leq M$) and the indices of the intermediate world coordinate and world coordinate arrays.

When the intermediate and world coordinate arrays contain the full complement of coordinate elements in image-order, as will usually be the case, then `map[m-1] == i-1` for axis `i` in the `N`-dimensional image ($1 \leq i \leq N$). In terms of the FITS keywords:

```
map[PVi_3a - 1] == i - 1.
```

However, a different association may result if the intermediate coordinates, for example, only contains a (relevant) subset of intermediate world coordinate elements. For example, if $M == 1$ for an image with $N > 1$, it is possible to fill the intermediate coordinates with the relevant coordinate element with `n_elem` set to 1. In this case `map[0] = 0` regardless of the value of i .

nc

`int` (read-only) Total number of coord vectors in the coord array.

Total number of coordinate vectors in the coordinate array being the product $K_1 * K_2 * \dots * K_M$.

p0

`int array[M]` Interpolated indices into the coordinate array.

Vector of length M of interpolated indices into the coordinate array such that `Upsilon_m`, as defined in Paper III, is equal to $(p0[m] + 1) + \text{delta}[m]$.

sense

`int array[M]` +1 if monotonically increasing, -1 if decreasing.

A vector of length M whose elements indicate whether the corresponding indexing vector is monotonically increasing (+1), or decreasing (-1).

Methods Documentation

print_contents ()

Print the contents of the `Tabprm` object to stdout. Probably only useful for debugging purposes, and may be removed in the future.

To get a string of the contents, use `repr`.

set ()

Allocates memory for work arrays.

Also sets up the class according to information supplied within it.

Note that this routine need not be called directly; it will be invoked by functions that need it.

Raises**MemoryError**

Memory allocation failed.

InvalidTabularParameters

Invalid tabular parameters.

WCS

```
class astropy.wcs.WCS (header=None, fobj=None, key='u', minerr=0.0, relax=True, naxis=None, key-  
sel=None, colsel=None, fix=True, translate_units='u', _do_set=True)
```

Bases: `astropy.wcs.WCSBase`

WCS objects perform standard WCS transformations, and correct for SIP and Paper IV table-lookup distortions, based on the WCS keywords and supplementary data read from a FITS file.

Parameters

header : `astropy.io.fits` header object, string, dict-like, or None, optional

If `header` is not provided or None, the object will be initialized to default values.

fobj : An `astropy.io.fits` file (`hdulist`) object, optional

It is needed when header keywords point to a [Paper IV](#) Lookup table distortion stored in a different extension.

key : str, optional

The name of a particular WCS transform to use. This may be either `' '` or `'A'-Z'` and corresponds to the "a" part of the `CTYPEia` cards. `key` may only be provided if `header` is also provided.

minerr : float, optional

The minimum value a distortion correction must have in order to be applied. If the value of `CQERRja` is smaller than `minerr`, the corresponding distortion is not applied.

relax : bool or int, optional

Degree of permissiveness:

- `True` (default): Admit all recognized informal extensions of the WCS standard.
- `False`: Recognize only FITS keywords defined by the published WCS standard.
- `int`: a bit field selecting specific extensions to accept. See [Header-reading relaxation constants](#) for details.

naxis : int or sequence, optional

Extracts specific coordinate axes using `sub()`. If a header is provided, and `naxis` is not `None`, `naxis` will be passed to `sub()` in order to select specific axes from the header. See `sub()` for more details about this parameter.

keysel : sequence of flags, optional

A sequence of flags used to select the keyword types considered by `wcslib`. When `None`, only the standard image header keywords are considered (and the underlying `wcspih()` C function is called). To use binary table image array or pixel list keywords, `keysel` must be set.

Each element in the list should be one of the following strings:

- `'image'`: Image header keywords
- `'binary'`: Binary table image array keywords
- `'pixel'`: Pixel list keywords

Keywords such as `EQUIna` or `RFRQna` that are common to binary table image arrays and pixel lists (including `WCSNna` and `TWCSna`) are selected by both `'binary'` and `'pixel'`.

colsel : sequence of int, optional

A sequence of table column numbers used to restrict the WCS transformations considered to only those pertaining to the specified columns. If `None`, there is no restriction.

fix : bool, optional

When `True` (default), call `fix` on the resulting object to fix any non-standard uses in the header. `FITSFixedWarning` Warnings will be emitted if any changes were made.

translate_units : str, optional

Specify which potentially unsafe translations of non-standard unit strings to perform. By default, performs none. See `WCS.fix` for more information about this parameter. Only effective when `fix` is `True`.

Raises

MemoryError

Memory allocation failed.

ValueError

Invalid key.

KeyError

Key not found in FITS header.

AssertionError

Lookup table distortion present in the header but *fobj* was not provided.

Notes

1. `astropy.wcs` supports arbitrary n dimensions for the core WCS (the transformations handled by WCSLIB). However, the Paper IV lookup table and SIP distortions must be two dimensional. Therefore, if you try to create a WCS object where the core WCS has a different number of dimensions than 2 and that object also contains a Paper IV lookup table or SIP distortion, a `ValueError` exception will be raised. To avoid this, consider using the *naxis* kwarg to select two dimensions from the core WCS.
2. The number of coordinate axes in the transformation is not determined directly from the `NAXIS` keyword but instead from the highest of:
 - `NAXIS` keyword
 - `WCSEXESa` keyword
 - The highest axis number in any parameterized WCS keyword. The keyvalue, as well as the keyword, must be syntactically valid otherwise it will not be considered.If none of these keyword types is present, i.e. if the header only contains auxiliary WCS keywords for a particular coordinate representation, then no coordinate description is constructed for it.

The number of axes, which is set as the *naxis* member, may differ for different coordinate representations of the same image.
3. When the header includes duplicate keywords, in most cases the last encountered is used.
4. `set` is called immediately after construction, so any invalid keywords or transformations will be raised by the constructor, not when subsequently calling a transformation method.

Attributes Summary

<code>axis_type_names</code>	World names for each coordinate axis
------------------------------	--------------------------------------

Methods Summary

<code>all_pix2world(*args, **kwargs)</code>	Transforms pixel coordinates to world coordinates.
---	--

Table 11.10 – continued from previous page

<code>all_world2pix(*args, **kwargs)</code>	Transforms world coordinates to pixel coordinates, using numerical iteration
<code>calcFootprint(*args, **kwargs)</code>	Deprecated since version 0.4.
<code>calc_footprint([header, undistort, axes, center])</code>	Calculates the footprint of the image on the sky.
<code>copy()</code>	Return a shallow copy of the object.
<code>deepcopy()</code>	Return a deep copy of the object.
<code>det2im(*args)</code>	Convert detector coordinates to image plane coordinates using Paper IV table
<code>dropaxis(dropax)</code>	Remove an axis from the WCS.
<code>fix([translate_units, naxis])</code>	Perform the fix operations from wcslib, and warn about any changes it has m
<code>footprint_to_file([filename, color, width])</code>	Writes out a ds9 style regions file.
<code>get_axis_types()</code>	Similar to <code>self.wcsprm.axis_types</code> but provides the information in a
<code>p4_pix2foc(*args)</code>	Convert pixel coordinates to focal plane coordinates using Paper IV table-look
<code>pix2foc(*args)</code>	Convert pixel coordinates to focal plane coordinates using the SIP polynomial
<code>printwcs()</code>	
<code>reorient_celestial_first()</code>	Reorient the WCS such that the celestial axes are first, followed by the spectr
<code>rotateCD(theta)</code>	
<code>sip_foc2pix(*args)</code>	Convert focal plane coordinates to pixel coordinates using the SIP polynomial
<code>sip_pix2foc(*args)</code>	Convert pixel coordinates to focal plane coordinates using the SIP polynomial
<code>slice(view[, numpy_order])</code>	Slice a WCS instance using a Numpy slice.
<code>sub(axes)</code>	Extracts the coordinate description for a subimage from a WCS object.
<code>swapaxes(ax0, ax1)</code>	Swap axes in a WCS.
<code>to_fits([relax, key])</code>	Generate an <code>astropy.io.fits.HDUList</code> object with all of the informa
<code>to_header([relax, key])</code>	Generate an <code>astropy.io.fits.Header</code> object with the basic WCS and
<code>to_header_string([relax])</code>	Identical to <code>to_header</code> , but returns a string containing the header cards.
<code>wcs_pix2world(*args, **kwargs)</code>	Transforms pixel coordinates to world coordinates by doing only the basic w
<code>wcs_world2pix(*args, **kwargs)</code>	Transforms world coordinates to pixel coordinates, using only the basic wcslib

Attributes Documentation

`axis_type_names`

World names for each coordinate axis

Returns

A list of names along each axis

Methods Documentation

`all_pix2world(*args, **kwargs)`

Transforms pixel coordinates to world coordinates.

Performs all of the following in order:

- Detector to image plane correction (optionally)
- [SIP](#) distortion correction (optionally)
- [Paper IV](#) table-lookup distortion correction (optionally)
- [wcslib](#) WCS transformation

Parameters

`args` : flexible

There are two accepted forms for the positional arguments:

- 2 arguments: An $N \times naxis$ array of coordinates, and an *origin*.

- more than 2 arguments: An array for each axis, followed by an *origin*. These arrays must be broadcastable to one another.

Here, *origin* is the coordinate in the upper left corner of the image. In FITS and Fortran standards, this is 1. In Numpy and C standards this is 0.

For a transformation that is not two-dimensional, the two-argument form must be used.

ra_dec_order : bool, optional

When `True` will ensure that world coordinates are always given and returned in as (*ra*, *dec*) pairs, regardless of the order of the axes specified by the in the `CTYPE` keywords. Default is `False`.

Returns

result : array

Returns the sky coordinates, in degrees. If the input was a single array and *origin*, a single array is returned, otherwise a tuple of arrays is returned.

Raises

MemoryError

Memory allocation failed.

SingularMatrixError

Linear transformation matrix is singular.

InconsistentAxisTypesError

Inconsistent or unrecognized coordinate axis types.

ValueError

Invalid parameter value.

ValueError

Invalid coordinate transformation parameters.

ValueError

x- and y-coordinate arrays are not the same size.

InvalidTransformError

Invalid coordinate transformation parameters.

InvalidTransformError

Ill-conditioned coordinate transformation parameters.

Notes

The order of the axes for the result is determined by the `CTYPEi` keywords in the FITS header, therefore it may not always be of the form (*ra*, *dec*). The `lat`, `lng`, `lattyp` and `lngtyp` members can be used to determine the order of the axes.

all_world2pix (*args, **kwargs)

Transforms world coordinates to pixel coordinates, using numerical iteration to invert the method `all_pix2world` within a tolerance of 1e-6 pixels.

Note that to use this function, you must have Scipy installed.

Parameters**args** : flexible

There are two accepted forms for the positional arguments:

- 2 arguments: An $N \times naxis$ array of coordinates, and an *origin*.
- more than 2 arguments: An array for each axis, followed by an *origin*. These arrays must be broadcastable to one another.

Here, *origin* is the coordinate in the upper left corner of the image. In FITS and Fortran standards, this is 1. In Numpy and C standards this is 0.

For a transformation that is not two-dimensional, the two-argument form must be used.

ra_dec_order : bool, optional

When `True` will ensure that world coordinates are always given and returned in as (*ra*, *dec*) pairs, regardless of the order of the axes specified by the in the `CTYPE` keywords. Default is `False`.

tolerance : float, optional

Tolerance of solution. Iteration terminates when the iterative solver estimates that the true solution is within this many pixels current estimate. Default value is 1e-6 (pixels).

Returns**result** : array

Returns the pixel coordinates. If the input was a single array and origin, a single array is returned, otherwise a tuple of arrays is returned.

Raises**MemoryError**

Memory allocation failed.

SingularMatrixError

Linear transformation matrix is singular.

InconsistentAxisTypesError

Inconsistent or unrecognized coordinate axis types.

ValueError

Invalid parameter value.

ValueError

Invalid coordinate transformation parameters.

ValueError

x- and y-coordinate arrays are not the same size.

InvalidTransformError

Invalid coordinate transformation parameters.

InvalidTransformError

Ill-conditioned coordinate transformation parameters.

Notes

The order of the axes for the input world array is determined by the `CTYPEi` keywords in the FITS header, therefore it may not always be of the form (ra, dec) . The `lat`, `lng`, `lattyp` and `lngtyp` members can be used to determine the order of the axes.

calcFootprint (**args, **kwargs*)

Deprecated since version 0.4: The `calcFootprint` function is deprecated and may be removed in a future version. Use `calc_footprint` instead.

calc_footprint (*header=None, undistort=True, axes=None, center=True*)

Calculates the footprint of the image on the sky.

A footprint is defined as the positions of the corners of the image on the sky after all available distortions have been applied.

Parameters

header : `Header` object, optional

undistort : bool, optional

If `True`, take SIP and distortion lookup table into account

axes : length 2 sequence ints, optional

If provided, use the given sequence as the shape of the image. Otherwise, use the `NAXIS1` and `NAXIS2` keywords from the header that was used to create this `WCS` object.

center : bool, optional

If `True` use the center of the pixel, otherwise use the corner.

Returns

coord : (4, 2) array of (x, y) coordinates.

The order is counter-clockwise starting with the bottom left corner.

copy ()

Return a shallow copy of the object.

Convenience method so user doesn't have to import the `copy` stdlib module.

deepcopy ()

Return a deep copy of the object.

Convenience method so user doesn't have to import the `copy` stdlib module.

det2im (**args*)

Convert detector coordinates to image plane coordinates using [Paper IV](#) table-lookup distortion correction.

The output is in absolute pixel coordinates, not relative to `CRPIX`.

Parameters

args : flexible

There are two accepted forms for the positional arguments:

- 2 arguments: An $N \times 2$ array of coordinates, and an *origin*.
- more than 2 arguments: An array for each axis, followed by an *origin*. These arrays must be broadcastable to one another.

Here, *origin* is the coordinate in the upper left corner of the image. In FITS and Fortran standards, this is 1. In Numpy and C standards this is 0.

Returns**result** : array

Returns the pixel coordinates. If the input was a single array and origin, a single array is returned, otherwise a tuple of arrays is returned.

Raises**MemoryError**

Memory allocation failed.

ValueError

Invalid coordinate transformation parameters.

dropaxis (*dropax*)

Remove an axis from the WCS.

Parameters**wcs** : `WCS`

The WCS with naxis to be chopped to naxis-1

dropax : int

The index of the WCS to drop, counting from 0 (i.e., python convention, not FITS convention)

Returns

A new `WCS` instance with one axis fewer

fix (*translate_units=u', naxis=None*)

Perform the fix operations from wcslib, and warn about any changes it has made.

Parameters**translate_units** : str, optional

Specify which potentially unsafe translations of non-standard unit strings to perform. By default, performs none.

Although "S" is commonly used to represent seconds, its translation to "s" is potentially unsafe since the standard recognizes "S" formally as Siemens, however rarely that may be used. The same applies to "H" for hours (Henry), and "D" for days (Debye).

This string controls what to do in such cases, and is case-insensitive.

- If the string contains "s", translate "S" to "s".
- If the string contains "h", translate "H" to "h".
- If the string contains "d", translate "D" to "d".

Thus '' doesn't do any unsafe translations, whereas 'shd' does all of them.

naxis : int array[naxis], optional

Image axis lengths. If this array is set to zero or `None`, then `cylfix` will not be invoked.

footprint_to_file (*filename=None, color=u'green', width=2*)

Writes out a ds9 style regions file. It can be loaded directly by ds9.

Parameters**filename** : str, optional

Output file name - default is 'footprint.reg'

color : str, optional

Color to use when plotting the line.

width : int, optional

Width of the region line.

get_axis_types ()

Similar to `self.wcsprm.axis_types` but provides the information in a more Python-friendly format.

Returns

result : list of dicts

Returns a list of dictionaries, one for each axis, each containing attributes about the type of that axis.

Each dictionary has the following keys:

- 'coordinate_type':
 - None: Non-specific coordinate type.
 - 'stokes': Stokes coordinate.
 - 'celestial': Celestial coordinate (including CUBEFACE).
 - 'spectral': Spectral coordinate.
- 'scale':
 - 'linear': Linear axis.
 - 'quantized': Quantized axis (STOKES, CUBEFACE).
 - 'non-linear celestial': Non-linear celestial axis.
 - 'non-linear spectral': Non-linear spectral axis.
 - 'logarithmic': Logarithmic axis.
 - 'tabular': Tabular axis.
- 'group'
 - Group number, e.g. lookup table number
- 'number'
 - For celestial axes:
 - *0: Longitude coordinate.
 - *1: Latitude coordinate.
 - *2: CUBEFACE number.
 - For lookup tables:
 - *the axis number in a multidimensional table.

CTYPEia in "4-3" form with unrecognized algorithm code will generate an error.

p4_pix2foc (*args)

Convert pixel coordinates to focal plane coordinates using [Paper IV](#) table-lookup distortion correction.

The output is in absolute pixel coordinates, not relative to CRPIX.

Parameters**args** : flexible

There are two accepted forms for the positional arguments:

- 2 arguments: An $N \times 2$ array of coordinates, and an *origin*.
- more than 2 arguments: An array for each axis, followed by an *origin*. These arrays must be broadcastable to one another.

Here, *origin* is the coordinate in the upper left corner of the image. In FITS and Fortran standards, this is 1. In Numpy and C standards this is 0.

Returns**result** : array

Returns the focal coordinates. If the input was a single array and origin, a single array is returned, otherwise a tuple of arrays is returned.

Raises**MemoryError**

Memory allocation failed.

ValueError

Invalid coordinate transformation parameters.

pix2foc (*args)

Convert pixel coordinates to focal plane coordinates using the [SIP](#) polynomial distortion convention and [Paper IV](#) table-lookup distortion correction.

The output is in absolute pixel coordinates, not relative to CRPIX.

Parameters**args** : flexible

There are two accepted forms for the positional arguments:

- 2 arguments: An $N \times 2$ array of coordinates, and an *origin*.
- more than 2 arguments: An array for each axis, followed by an *origin*. These arrays must be broadcastable to one another.

Here, *origin* is the coordinate in the upper left corner of the image. In FITS and Fortran standards, this is 1. In Numpy and C standards this is 0.

Returns**result** : array

Returns the focal coordinates. If the input was a single array and origin, a single array is returned, otherwise a tuple of arrays is returned.

Raises**MemoryError**

Memory allocation failed.

ValueError

Invalid coordinate transformation parameters.

printwcs ()

reorient_celestial_first()

Reorient the WCS such that the celestial axes are first, followed by the spectral axis, followed by any others. Assumes at least celestial axes are present.

rotateCD(*theta*)**sip_foc2pix**(*args)

Convert focal plane coordinates to pixel coordinates using the SIP polynomial distortion convention.

Paper IV table lookup distortion correction is not applied, even if that information existed in the FITS file that initialized this WCS object.

Parameters

args : flexible

There are two accepted forms for the positional arguments:

- 2 arguments: An $N \times 2$ array of coordinates, and an *origin*.
- more than 2 arguments: An array for each axis, followed by an *origin*. These arrays must be broadcastable to one another.

Here, *origin* is the coordinate in the upper left corner of the image. In FITS and Fortran standards, this is 1. In Numpy and C standards this is 0.

Returns

result : array

Returns the pixel coordinates. If the input was a single array and origin, a single array is returned, otherwise a tuple of arrays is returned.

Raises**MemoryError**

Memory allocation failed.

ValueError

Invalid coordinate transformation parameters.

sip_pix2foc(*args)

Convert pixel coordinates to focal plane coordinates using the SIP polynomial distortion convention.

The output is in pixel coordinates, relative to CRPIX.

Paper IV table lookup distortion correction is not applied, even if that information existed in the FITS file that initialized this WCS object. To correct for that, use `pix2foc` or `p4_pix2foc`.

Parameters

args : flexible

There are two accepted forms for the positional arguments:

- 2 arguments: An $N \times 2$ array of coordinates, and an *origin*.
- more than 2 arguments: An array for each axis, followed by an *origin*. These arrays must be broadcastable to one another.

Here, *origin* is the coordinate in the upper left corner of the image. In FITS and Fortran standards, this is 1. In Numpy and C standards this is 0.

Returns

result : array

Returns the focal coordinates. If the input was a single array and origin, a single array is returned, otherwise a tuple of arrays is returned.

Raises

MemoryError

Memory allocation failed.

ValueError

Invalid coordinate transformation parameters.

slice (*view*, *numpy_order=True*)

Slice a WCS instance using a Numpy slice. The order of the slice should be reversed (as for the data) compared to the natural WCS order.

Parameters

view : tuple

A tuple containing the same number of slices as the WCS system. The `step` method, the third argument to a slice, is not presently supported.

numpy_order : bool

Use numpy order, i.e. slice the WCS so that an identical slice applied to a numpy array will slice the array and WCS in the same way. If set to `False`, the WCS will be sliced in FITS order, meaning the first slice will be applied to the *last* numpy index but the *first* WCS axis.

Returns

wcs_new : `WCS`

A new resampled WCS axis

sub (*axes*)

Extracts the coordinate description for a subimage from a `WCS` object.

The world coordinate system of the subimage must be separable in the sense that the world coordinates at any point in the subimage must depend only on the pixel coordinates of the axes extracted. In practice, this means that the `PCi_ja` matrix of the original image must not contain non-zero off-diagonal terms that associate any of the subimage axes with any of the non-subimage axes.

`sub` can also add axes to a `wcsprm` object. The new axes will be created using the defaults set by the `Wcsprm` constructor which produce a simple, unnamed, linear axis with world coordinates equal to the pixel coordinate. These default values can be changed before invoking `set`.

Parameters

axes : int or a sequence.

- If an int, include the first N axes in their original order.
- If a sequence, may contain a combination of image axis numbers (1-relative) or special axis identifiers (see below). Order is significant; `axes[0]` is the axis number of the input image that corresponds to the first axis in the subimage, etc. Use an axis number of 0 to create a new axis using the defaults.
- If 0, [] or `None`, do a deep copy.

Coordinate axes types may be specified using either strings or special integer constants. The available types are:

- `'longitude'` / `WCSSUB_LONGITUDE`: Celestial longitude
- `'latitude'` / `WCSSUB_LATITUDE`: Celestial latitude

- 'cubeface' / WCSSUB_CUBEFACE: Quadcube CUBEFACE axis
- 'spectral' / WCSSUB_SPECTRAL: Spectral axis
- 'stokes' / WCSSUB_STOKES: Stokes axis
- 'celestial' / WCSSUB_CELESTIAL: An alias for the combination of 'longitude', 'latitude' and 'cubeface'.

Returns

`new_wcs` : WCS object

Raises**MemoryError**

Memory allocation failed.

InvalidSubimageSpecificationError

Invalid subimage specification (no spectral axis).

NonseparableSubimageCoordinateSystem

Non-separable subimage coordinate system.

Notes

Combinations of subimage axes of particular types may be extracted in the same order as they occur in the input image by combining the integer constants with the 'binary or' (`|`) operator. For example:

```
wcs.sub([WCSSUB_LONGITUDE | WCSSUB_LATITUDE | WCSSUB_SPECTRAL])
```

would extract the longitude, latitude, and spectral axes in the same order as the input image. If one of each were present, the resulting object would have three dimensions.

For convenience, `WCSSUB_CELESTIAL` is defined as the combination `WCSSUB_LONGITUDE | WCSSUB_LATITUDE | WCSSUB_CUBEFACE`.

The codes may also be negated to extract all but the types specified, for example:

```
wcs.sub([
    WCSSUB_LONGITUDE,
    WCSSUB_LATITUDE,
    WCSSUB_CUBEFACE,
    -(WCSSUB_SPECTRAL | WCSSUB_STOKES)])
```

The last of these specifies all axis types other than spectral or Stokes. Extraction is done in the order specified by `axes`, i.e. a longitude axis (if present) would be extracted first (via `axes[0]`) and not subsequently (via `axes[3]`). Likewise for the latitude and cubeface axes in this example.

The number of dimensions in the returned object may be less than or greater than the length of `axes`. However, it will never exceed the number of axes in the input image.

swapaxes (*ax0*, *ax1*)

Swap axes in a WCS.

Parameters

`wcs`: '~astropy.wcs.WCS'

The WCS to have its axes swapped

ax0: int

ax1: int

The indices of the WCS to be swapped, counting from 0 (i.e., python convention, not FITS convention)

Returns

A new `WCS` instance with the same number of axes, but two swapped

to_fits (*relax=False, key=None*)

Generate an `astropy.io.fits.HDUList` object with all of the information stored in this object. This should be logically identical to the input FITS file, but it will be normalized in a number of ways.

See `to_header` for some warnings about the output produced.

Parameters

relax : bool or int, optional

Degree of permissiveness:

- `False` (default): Write all extensions that are considered to be safe and recommended.
- `True`: Write all recognized informal extensions of the WCS standard.
- `int`: a bit field selecting specific extensions to write. See *Header-writing relaxation constants* for details.

key : str

The name of a particular WCS transform to use. This may be either ' ' or 'A'-Z' and corresponds to the "a" part of the `CTYPEia` cards.

Returns

hdulist : `astropy.io.fits.HDUList`

to_header (*relax=False, key=None*)

Generate an `astropy.io.fits.Header` object with the basic WCS and SIP information stored in this object. This should be logically identical to the input FITS file, but it will be normalized in a number of ways.

Warning: This function does not write out Paper IV distortion information, since that requires multiple FITS header data units. To get a full representation of everything in this object, use `to_fits`.

Parameters

relax : bool or int, optional

Degree of permissiveness:

- `False` (default): Write all extensions that are considered to be safe and recommended.
- `True`: Write all recognized informal extensions of the WCS standard.
- `int`: a bit field selecting specific extensions to write. See *Header-writing relaxation constants* for details.

key : str

The name of a particular WCS transform to use. This may be either ' ' or 'A'-Z' and corresponds to the "a" part of the `CTYPEia` cards.

Returns

header : `astropy.io.fits.Header`

Notes

The output header will almost certainly differ from the input in a number of respects:

- 1.The output header only contains WCS-related keywords. In particular, it does not contain syntactically-required keywords such as `SIMPLE`, `NAXIS`, `BITPIX`, or `END`.
- 2.Deprecated (e.g. `CROTAN`) or non-standard usage will be translated to standard (this is partially dependent on whether `fix` was applied).
- 3.Quantities will be converted to the units used internally, basically SI with the addition of degrees.
- 4.Floating-point quantities may be given to a different decimal precision.
- 5.Elements of the `PCi_j` matrix will be written if and only if they differ from the unit matrix. Thus, if the matrix is unity then no elements will be written.
- 6.Additional keywords such as `WCSAXES`, `CUNITi`, `LONPOLEa` and `LATPOLEa` may appear.
- 7.The original keycomments will be lost, although `to_header` tries hard to write meaningful comments.
- 8.Keyword order may be changed.

to_header_string (*relax=False*)

Identical to `to_header`, but returns a string containing the header cards.

wcs_pix2world (**args, **kwargs*)

Transforms pixel coordinates to world coordinates by doing only the basic `wcslib` transformation.

No `SIP` or `Paper IV` table lookup distortion correction is applied. To perform distortion correction, see `all_pix2world`, `sip_pix2foc`, `p4_pix2foc`, or `pix2foc`.

Parameters

args : flexible

There are two accepted forms for the positional arguments:

- 2 arguments: An $N \times naxis$ array of coordinates, and an *origin*.
- more than 2 arguments: An array for each axis, followed by an *origin*. These arrays must be broadcastable to one another.

Here, *origin* is the coordinate in the upper left corner of the image. In FITS and Fortran standards, this is 1. In Numpy and C standards this is 0.

For a transformation that is not two-dimensional, the two-argument form must be used.

ra_dec_order : bool, optional

When `True` will ensure that world coordinates are always given and returned in as (*ra*, *dec*) pairs, regardless of the order of the axes specified by the in the `CTYPE` keywords. Default is `False`.

Returns

result : array

Returns the world coordinates, in degrees. If the input was a single array and *origin*, a single array is returned, otherwise a tuple of arrays is returned.

Raises**MemoryError**

Memory allocation failed.

SingularMatrixError

Linear transformation matrix is singular.

InconsistentAxisTypesError

Inconsistent or unrecognized coordinate axis types.

ValueError

Invalid parameter value.

ValueError

Invalid coordinate transformation parameters.

ValueError

x- and y-coordinate arrays are not the same size.

InvalidTransformError

Invalid coordinate transformation parameters.

InvalidTransformError

Ill-conditioned coordinate transformation parameters.

Notes

The order of the axes for the result is determined by the `CTYPEi` keywords in the FITS header, therefore it may not always be of the form (ra, dec) . The `lat`, `lng`, `lattyp` and `lngtyp` members can be used to determine the order of the axes.

wcs_world2pix (*args, **kwargs)

Transforms world coordinates to pixel coordinates, using only the basic `wcslib` WCS transformation. No SIP or Paper IV table lookup distortion is applied.

Parameters

args : flexible

There are two accepted forms for the positional arguments:

- 2 arguments: An $N \times naxis$ array of coordinates, and an *origin*.
- more than 2 arguments: An array for each axis, followed by an *origin*. These arrays must be broadcastable to one another.

Here, *origin* is the coordinate in the upper left corner of the image. In FITS and Fortran standards, this is 1. In Numpy and C standards this is 0.

For a transformation that is not two-dimensional, the two-argument form must be used.

ra_dec_order : bool, optional

When `True` will ensure that world coordinates are always given and returned in as (ra, dec) pairs, regardless of the order of the axes specified by the in the `CTYPE` keywords. Default is `False`.

Returns**result** : array

Returns the pixel coordinates. If the input was a single array and origin, a single array is returned, otherwise a tuple of arrays is returned.

Raises**MemoryError**

Memory allocation failed.

SingularMatrixError

Linear transformation matrix is singular.

InconsistentAxisTypesError

Inconsistent or unrecognized coordinate axis types.

ValueError

Invalid parameter value.

ValueError

Invalid coordinate transformation parameters.

ValueError

x- and y-coordinate arrays are not the same size.

InvalidTransformError

Invalid coordinate transformation parameters.

InvalidTransformError

Ill-conditioned coordinate transformation parameters.

Notes

The order of the axes for the input world array is determined by the `CTYPEi` keywords in the FITS header, therefore it may not always be of the form *(ra, dec)*. The `lat`, `lng`, `lattyp` and `lngtyp` members can be used to determine the order of the axes.

WCSBase**class** `astropy.wcs.WCSBase`Bases: `object`

Wcs objects amalgamate basic WCS (as provided by `wcslib`), with `SIP` and `Paper IV` distortion operations.

To perform all distortion corrections and WCS transformation, use `all_pix2world`.

Parameters**sip** : `Sip` object or `None`**cpdis** : A pair of `DistortionLookupTable` objects, or

(`None`, `None`).

wcsprm : `Wcsprm` object**det2im** : A pair of `DistortionLookupTable` objects, or

(None, None).

Attributes Summary

<code>cpdis1</code>	<code>DistortionLookupTable</code>
<code>cpdis2</code>	<code>DistortionLookupTable</code>
<code>det2im1</code>	A <code>DistortionLookupTable</code> object for detector to image plane correction in the <i>x</i> -axis.
<code>det2im2</code>	A <code>DistortionLookupTable</code> object for detector to image plane correction in the <i>y</i> -axis.
<code>sip</code>	Get/set the <code>Sip</code> object for performing <code>SIP</code> distortion correction.
<code>wcs</code>	A <code>Wcsprm</code> object to perform the basic <code>wcslib</code> WCS transformation.

Attributes Documentation

`cpdis1`

`DistortionLookupTable`

The pre-linear transformation distortion lookup table, CPDIS1.

`cpdis2`

`DistortionLookupTable`

The pre-linear transformation distortion lookup table, CPDIS2.

`det2im1`

A `DistortionLookupTable` object for detector to image plane correction in the *x*-axis.

`det2im2`

A `DistortionLookupTable` object for detector to image plane correction in the *y*-axis.

`sip`

Get/set the `Sip` object for performing `SIP` distortion correction.

`wcs`

A `Wcsprm` object to perform the basic `wcslib` WCS transformation.

`WcsError`

exception `astropy.wcs.WcsError`

Base class of all invalid WCS errors.

`Wcsprm`

class `astropy.wcs.Wcsprm`

Bases: `object`

`Wcsprm` is a direct wrapper around `wcslib`. It provides access to the core WCS transformations that it supports.

Note: The members of this object correspond roughly to the key/value pairs in the FITS header. However, they are adjusted and normalized in a number of ways that make performing the WCS transformation easier. Therefore, they can not be relied upon to get the original values in the header. For that, use `astropy.io.fits.Header` directly.

The FITS header parsing enforces correct FITS “keyword = value” syntax with regard to the equals sign occurring in columns 9 and 10. However, it does recognize free-format character (NOST 100-2.0, Sect. 5.2.1), integer (Sect. 5.2.3), and floating-point values (Sect. 5.2.4) for all keywords.

Parameters

header : An `astropy.io.fits.Header`, string, or `None`.

If `None`, the object will be initialized to default values.

key : str, optional

The key referring to a particular WCS transform in the header. This may be either `'` or `'A'-'Z'` and corresponds to the `"a"` part of `"CTYPEia"`. (*key* may only be provided if *header* is also provided.)

relax : bool or int, optional

Degree of permissiveness:

- `False`: Recognize only FITS keywords defined by the published WCS standard.
- `True`: Admit all recognized informal extensions of the WCS standard.
- `int`: a bit field selecting specific extensions to accept. See *Header-reading relaxation constants* for details.

naxis : int, optional

The number of world coordinates axes for the object. (*naxis* may only be provided if *header* is `None`.)

keysel : sequence of flag bits, optional

Vector of flag bits that may be used to restrict the keyword types considered:

- `WCSHDR_IMGHEAD`: Image header keywords.
- `WCSHDR_BIMGARR`: Binary table image array.
- `WCSHDR_PIXLIST`: Pixel list keywords.

If zero, there is no restriction. If -1, the underlying wcslib function `wcspih()` is called, rather than `wcstbh()`.

colsel : sequence of int

A sequence of table column numbers used to restrict the keywords considered. `None` indicates no restriction.

Raises**MemoryError**

Memory allocation failed.

ValueError

Invalid key.

KeyError

Key not found in FITS header.

Continued on next page

Table 11.12 – continued from previous page

Attributes Summary

<code>alt</code>	<code>str</code> Character code for alternate coordinate descriptions.
<code>axis_types</code>	<code>int array[naxis]</code> An array of four-digit type codes for each axis.
<code>cd</code>	<code>double array[naxis][naxis]</code> The <code>CDi_ja</code> linear transformation
<code>cdelt</code>	<code>double array[naxis]</code> Coordinate increments (<code>CDELTi</code>) for each
<code>cel_offset</code>	<code>boolean</code> Is there an offset?
<code>cname</code>	<code>list of strings</code> A list of the coordinate axis names, from
<code>colax</code>	<code>int array[naxis]</code> An array recording the column numbers for each
<code>colnum</code>	<code>int</code> Column of FITS binary table associated with this WCS.
<code>crder</code>	<code>double array[naxis]</code> The random error in each coordinate axis,
<code>crota</code>	<code>double array[naxis]</code> <code>CROTAi</code> keyvalues for each coordinate
<code>crpix</code>	<code>double array[naxis]</code> Coordinate reference pixels (<code>CRPIXj</code>) for
<code>crval</code>	<code>double array[naxis]</code> Coordinate reference values (<code>CRVALi</code>) for
<code>csyer</code>	<code>double array[naxis]</code> The systematic error in the coordinate value
<code>ctype</code>	<code>list of strings[naxis]</code> List of <code>CTYPEi</code> keyvalues.
<code>cubeface</code>	<code>int</code> Index into the <code>pixcrd</code> (pixel coordinate) array for the
<code>cunit</code>	<code>list of astropy.UnitBase[naxis]</code> List of <code>CUNITi</code> keyvalues as
<code>dateavg</code>	<code>string</code> Representative mid-point of the date of observation.
<code>dateobs</code>	<code>string</code> Start of the date of observation.
<code>equinox</code>	<code>double</code> The equinox associated with dynamical equatorial or
<code>imgpix_matrix</code>	<code>double array[2][2]</code> (read-only) Inverse of the <code>CDELTA</code> or <code>PC</code>
<code>lat</code>	<code>int</code> (read-only) The index into the world coord array containing
<code>latpole</code>	<code>double</code> The native latitude of the celestial pole, <code>LATPOLEa</code> (deg).
<code>latty</code>	<code>string</code> (read-only) Celestial axis type for latitude.
<code>lng</code>	<code>int</code> (read-only) The index into the world coord array containing
<code>lngtyp</code>	<code>string</code> (read-only) Celestial axis type for longitude.
<code>lonpole</code>	<code>double</code> The native longitude of the celestial pole.
<code>mjdavg</code>	<code>double</code> Modified Julian Date corresponding to <code>DATE-AVG</code> .
<code>mjdobs</code>	<code>double</code> Modified Julian Date corresponding to <code>DATE-OBS</code> .
<code>name</code>	<code>string</code> The name given to the coordinate representation
<code>naxis</code>	<code>int</code> (read-only) The number of axes (pixel and coordinate).
<code>obsgeo</code>	<code>double array[3]</code> Location of the observer in a standard terrestrial
<code>pc</code>	<code>double array[naxis][naxis]</code> The <code>PCi_ja</code> (pixel coordinate)
<code>phi0</code>	<code>double</code> The native latitude of the fiducial point.
<code>piximg_matrix</code>	<code>double array[2][2]</code> (read-only) Matrix containing the product of
<code>radesys</code>	<code>string</code> The equatorial or ecliptic coordinate system type,
<code>restfrq</code>	<code>double</code> Rest frequency (Hz) from <code>RESTFRQa</code> .
<code>restwav</code>	<code>double</code> Rest wavelength (m) from <code>RESTWAVa</code> .
<code>spec</code>	<code>int</code> (read-only) The index containing the spectral axis values.
<code>specsys</code>	<code>string</code> Spectral reference frame (standard of rest), <code>SPECSYSa</code> .
<code>ssysobs</code>	<code>string</code> Spectral reference frame.
<code>ssyssrc</code>	<code>string</code> Spectral reference frame for redshift.
<code>tab</code>	<code>list of Tabprm</code> Tabular coordinate objects.
<code>theta0</code>	<code>double</code> The native longitude of the fiducial point.
<code>velangl</code>	<code>double</code> Velocity angle.
<code>velosys</code>	<code>double</code> Relative radial velocity.
<code>zsource</code>	<code>double</code> The redshift, <code>ZSOURCEa</code> , of the source.

Methods Summary

<code>bounds_check(pix2world, world2pix)</code>	Enable/disable bounds checking.
<code>cdfix()</code>	Fix erroneously omitted <code>CDi_ja</code> keywords.
<code>celfix</code>	Translates AIPS-convention celestial projection types, <code>-NCP</code> and <code>-GLS</code> .
<code>compare(other[, cmp])</code>	Compare two <code>Wcsprm</code> objects for equality.
<code>cylfix()</code>	Fixes WCS keyvalues for malformed cylindrical projections.
<code>datfix()</code>	Translates the old <code>DATE-OBS</code> date format to year-2000 standard form (<code>yyyy-mm-dd</code>).
<code>fix([translate_units, naxis])</code>	Applies all of the corrections handled separately by <code>datfix</code> , <code>unitfix</code> , <code>celfix</code> , <code>sp</code> .
<code>get_cdelmt()</code> -> double array[naxis]	Coordinate increments (<code>CDELTi</code>) for each coord axis.
<code>get_pc()</code> -> double array[naxis][naxis]	Returns the PC matrix in read-only form.
<code>get_ps()</code> -> list of tuples	Returns <code>PSi_ma</code> keywords for each <i>i</i> and <i>m</i> .
<code>get_pv()</code> -> list of tuples	Returns <code>PVi_ma</code> keywords for each <i>i</i> and <i>m</i> .
<code>has_cd()</code> -> bool	Returns <code>True</code> if <code>CDi_ja</code> is present.
<code>has_cdi_ja()</code> -> bool	Alias for <code>has_cd</code> .
<code>has_crota()</code> -> bool	Returns <code>True</code> if <code>CROTAi</code> is present.
<code>has_crotaia()</code> -> bool	Alias for <code>has_crota</code> .
<code>has_pc()</code> -> bool	Returns <code>True</code> if <code>PCi_ja</code> is present.
<code>has_pci_ja()</code> -> bool	Alias for <code>has_pc</code> .
<code>is_unity()</code> -> bool	Returns <code>True</code> if the linear transformation matrix (<code>cd</code>) is unity.
<code>mix(mixpix, mixcel, vspan, vstep, viter, ...)</code>	Given either the celestial longitude or latitude plus an element of the pixel coordinate, <code>s</code> .
<code>p2s(pixcrd, origin)</code>	Converts pixel to world coordinates.
<code>print_contents()</code>	Print the contents of the <code>Wcsprm</code> object to stdout.
<code>s2p(world, origin)</code>	Transforms world coordinates to pixel coordinates.
<code>set()</code>	Sets up a WCS object for use according to information supplied within it.
<code>set_ps(ps)</code>	Sets <code>PSi_ma</code> keywords for each <i>i</i> and <i>m</i> .
<code>set_pv(pv)</code>	Sets <code>PVi_ma</code> keywords for each <i>i</i> and <i>m</i> .
<code>spcfix()</code> -> int	Translates AIPS-convention spectral coordinate types.
<code>sptr ctype[, i]</code>	Translates the spectral axis in a WCS object.
<code>sub(axes)</code>	Extracts the coordinate description for a subimage from a <code>WCS</code> object.
<code>to_header([relax])</code>	<code>to_header</code> translates a WCS object into a FITS header.
<code>unitfix([translate_units])</code>	Translates non-standard <code>CUNITi</code> keyvalues.

Attributes Documentation

alt

`str` Character code for alternate coordinate descriptions.

For example, the "a" in keyword names such as `CTYPEia`. This is a space character for the primary coordinate description, or one of the 26 upper-case letters, A-Z.

axis_types

`int array[naxis]` An array of four-digit type codes for each axis.

- First digit (i.e. 1000s):
 - 0: Non-specific coordinate type.
 - 1: Stokes coordinate.
 - 2: Celestial coordinate (including `CUBEFACE`).
 - 3: Spectral coordinate.
- Second digit (i.e. 100s):
 - 0: Linear axis.

- 1: Quantized axis (STOKES, CUBEFACE).
- 2: Non-linear celestial axis.
- 3: Non-linear spectral axis.
- 4: Logarithmic axis.
- 5: Tabular axis.
- Third digit (i.e. 10s):
 - 0: Group number, e.g. lookup table number
- The fourth digit is used as a qualifier depending on the axis type.
 - For celestial axes:
 - *0: Longitude coordinate.
 - *1: Latitude coordinate.
 - *2: CUBEFACE number.
 - For lookup tables: the axis number in a multidimensional table.

CTYPE_ia in "4-3" form with unrecognized algorithm code will have its type set to -1 and generate an error.

cd

double array[naxis][naxis] The CD_i_ja linear transformation matrix.

For historical compatibility, three alternate specifications of the linear transformations are available in wcslib. The canonical PC_i_ja with CDELT_ia, CD_i_ja, and the deprecated CROTA_ia keywords. Although the latter may not formally co-exist with PC_i_ja, the approach here is simply to ignore them if given in conjunction with PC_i_ja.

has_pc, has_cd and has_crota can be used to determine which of these alternatives are present in the header.

These alternate specifications of the linear transformation matrix are translated immediately to PC_i_ja by set and are nowhere visible to the lower-level routines. In particular, set resets cdelt to unity if CD_i_ja is present (and no PC_i_ja). If no CROTA_ia is associated with the latitude axis, set reverts to a unity PC_i_ja matrix.

cdelt

double array[naxis] Coordinate increments (CDELT_ia) for each coord axis.

If a CD_i_ja linear transformation matrix is present, a warning is raised and cdelt is ignored. The CD_i_ja matrix may be deleted by:

```
del wcs.wcs.cd
```

An undefined value is represented by NaN.

cel_offset

boolean Is there an offset?

If True, an offset will be applied to (x, y) to force (x, y) = (0, 0) at the fiducial point, (phi_0, theta_0). Default is False.

cname

list of strings A list of the coordinate axis names, from CNAME_ia.

colax

int array[naxis] An array recording the column numbers for each axis in a pixel list.

colnum

`int` Column of FITS binary table associated with this WCS.

Where the coordinate representation is associated with an image-array column in a FITS binary table, this property may be used to record the relevant column number.

It should be set to zero for an image header or pixel list.

order

`double array[naxis]` The random error in each coordinate axis, `CRDERia`.

An undefined value is represented by NaN.

crota

`double array[naxis]` `CROTAia` keyvalues for each coordinate axis.

For historical compatibility, three alternate specifications of the linear transformations are available in `wcslib`. The canonical `PCi_ja` with `CDELTia`, `CDi_ja`, and the deprecated `CROTAia` keywords. Although the latter may not formally co-exist with `PCi_ja`, the approach here is simply to ignore them if given in conjunction with `PCi_ja`.

`has_pc`, `has_cd` and `has_crota` can be used to determine which of these alternatives are present in the header.

These alternate specifications of the linear transformation matrix are translated immediately to `PCi_ja` by `set` and are nowhere visible to the lower-level routines. In particular, `set` resets `cdelt` to unity if `CDi_ja` is present (and no `PCi_ja`). If no `CROTAia` is associated with the latitude axis, `set` reverts to a unity `PCi_ja` matrix.

crpix

`double array[naxis]` Coordinate reference pixels (`CRPIXja`) for each pixel axis.

crval

`double array[naxis]` Coordinate reference values (`CRVALia`) for each coordinate axis.

csyer

`double array[naxis]` The systematic error in the coordinate value axes, `CSYERia`.

An undefined value is represented by NaN.

ctype

`list of strings[naxis]` List of `CTYPEia` keyvalues.

The `ctype` keyword values must be in upper case and there must be zero or one pair of matched celestial axis types, and zero or one spectral axis.

cubeface

`int` Index into the `pixcrd` (pixel coordinate) array for the `CUBEFACE` axis.

This is used for quadcube projections where the cube faces are stored on a separate axis.

The quadcube projections (`TSC`, `CSC`, `QSC`) may be represented in FITS in either of two ways:

- The six faces may be laid out in one plane and numbered as follows:

```
0
4 3 2 1 4 3 2
5
```

Faces 2, 3 and 4 may appear on one side or the other (or both). The world-to-pixel routines map faces 2, 3 and 4 to the left but the pixel-to-world routines accept them on either side.

- The COBE convention in which the six faces are stored in a three-dimensional structure using a CUBEFACE axis indexed from 0 to 5 as above.

These routines support both methods; `set` determines which is being used by the presence or absence of a CUBEFACE axis in `ctype`. `p2s` and `s2p` translate the CUBEFACE axis representation to the single plane representation understood by the lower-level projection routines.

cunit

list of `astropy.UnitBase[naxis]` List of CUNITia keyvalues as `astropy.units.UnitBase` instances.

These define the units of measurement of the `CRVALia`, `CDELTAia` and `CDI_ja` keywords.

As CUNITia is an optional header keyword, `cunit` may be left blank but otherwise is expected to contain a standard units specification as defined by WCS Paper I. `unitfix` is available to translate commonly used non-standard units specifications but this must be done as a separate step before invoking `set`.

For celestial axes, if `cunit` is not blank, `set` uses `wcsunits` to parse it and scale `cdelt`, `crval`, and `cd` to decimal degrees. It then resets `cunit` to "deg".

For spectral axes, if `cunit` is not blank, `set` uses `wcsunits` to parse it and scale `cdelt`, `crval`, and `cd` to SI units. It then resets `cunit` accordingly.

`set` ignores `cunit` for other coordinate types; `cunit` may be used to label coordinate values.

dateavg

string Representative mid-point of the date of observation.

In ISO format, `yyyy-mm-ddThh:mm:ss`.

See also:

`astropy.wcs.Wcsprm.dateobs`

dateobs

string Start of the date of observation.

In ISO format, `yyyy-mm-ddThh:mm:ss`.

See also:

`astropy.wcs.Wcsprm.dateavg`

equinox

double The equinox associated with dynamical equatorial or ecliptic coordinate systems.

EQUINOXa (or EPOCH in older headers). Not applicable to ICRS equatorial or ecliptic coordinates.

An undefined value is represented by NaN.

imgpix_matrix

double array[2][2] (read-only) Inverse of the CDELTA or PC matrix.

Inverse containing the product of the CDELTAia diagonal matrix and the PCi_ja matrix.

lat

int (read-only) The index into the world coord array containing latitude values.

latpole

double The native latitude of the celestial pole, LATPOLEa (deg).

lattyp

string (read-only) Celestial axis type for latitude.

For example, "RA", "DEC", "GLON", "GLAT", etc. extracted from "RA-", "DEC-", "GLON", "GLAT", etc. in the first four characters of CTYPEia but with trailing dashes removed.

l_{ng}

`int` (read-only) The index into the world coord array containing longitude values.

l_{ngtyp}

`string` (read-only) Celestial axis type for longitude.

For example, “RA”, “DEC”, “GLON”, “GLAT”, etc. extracted from “RA-”, “DEC-”, “GLON”, “GLAT”, etc. in the first four characters of `CTYPEia` but with trailing dashes removed.

l_{onpole}

`double` The native longitude of the celestial pole.

`LONPOLEa` (deg).

m_{jdavg}

`double` Modified Julian Date corresponding to `DATE-AVG`.

($MJD = JD - 2400000.5$).

An undefined value is represented by NaN.

See also:

`astropy.wcs.Wcsprm.mjdobs`

m_{jdobs}

`double` Modified Julian Date corresponding to `DATE-OBS`.

($MJD = JD - 2400000.5$).

An undefined value is represented by NaN.

See also:

`astropy.wcs.Wcsprm.mjdavg`

name

`string` The name given to the coordinate representation `WCSNAMEa`.

n_{axis}

`int` (read-only) The number of axes (pixel and coordinate).

Given by the `NAXIS` or `WCSAXESa` keyvalues.

The number of coordinate axes is determined at parsing time, and can not be subsequently changed.

It is determined from the highest of the following:

1.`NAXIS`

2.`WCSAXESa`

3.The highest axis number in any parameterized WCS keyword. The keyvalue, as well as the keyword, must be syntactically valid otherwise it will not be considered.

If none of these keyword types is present, i.e. if the header only contains auxiliary WCS keywords for a particular coordinate representation, then no coordinate description is constructed for it.

This value may differ for different coordinate representations of the same image.

obsgeo

`double array[3]` Location of the observer in a standard terrestrial reference frame.

`OBSGEO-X`, `OBSGEO-Y`, `OBSGEO-Z` (in meters).

An undefined value is represented by NaN.

pc

double array[naxis][naxis] The PCi_ja (pixel coordinate) transformation matrix.

The order is:

```
[[PC1_1, PC1_2],
 [PC2_1, PC2_2]]
```

For historical compatibility, three alternate specifications of the linear transformations are available in wcslib. The canonical PCi_ja with CDELTia, CDi_ja, and the deprecated CROTAia keywords. Although the latter may not formally co-exist with PCi_ja, the approach here is simply to ignore them if given in conjunction with PCi_ja.

has_pc, has_cd and has_crota can be used to determine which of these alternatives are present in the header.

These alternate specifications of the linear transformation matrix are translated immediately to PCi_ja by set and are nowhere visible to the lower-level routines. In particular, set resets cdelt to unity if CDi_ja is present (and no PCi_ja). If no CROTAia is associated with the latitude axis, set reverts to a unity PCi_ja matrix.

phi0

double The native latitude of the fiducial point.

The point whose celestial coordinates are given in ref[1:2]. If undefined (NaN) the initialization routine, set, will set this to a projection-specific default.

See also:

`astropy.wcs.Wcsprm.theta0`

piximg_matrix

double array[2][2] (read-only) Matrix containing the product of the CDELTia diagonal matrix and the PCi_ja matrix.

radesys

string The equatorial or ecliptic coordinate system type, RADESYSa.

restfrq

double Rest frequency (Hz) from RESTFRQa.

An undefined value is represented by NaN.

restwav

double Rest wavelength (m) from RESTWAVa.

An undefined value is represented by NaN.

spec

int (read-only) The index containing the spectral axis values.

specsyst

string Spectral reference frame (standard of rest), SPECSYSa.

See also:

`astropy.wcs.Wcsprm.ssysobs`, `astropy.wcs.Wcsprm.velosys`

ssysobs

string Spectral reference frame.

The spectral reference frame in which there is no differential variation in the spectral coordinate across the field-of-view, SSYSOBSa.

See also:

`astropy.wcs.Wcsprm.specsys`, `astropy.wcs.Wcsprm.velosys`

ssysrc

string Spectral reference frame for redshift.

The spectral reference frame (standard of rest) in which the redshift was measured, SSYSSRCa.

tab

list of Tabprm Tabular coordinate objects.

A list of tabular coordinate objects associated with this WCS.

theta0

double The native longitude of the fiducial point.

The point whose celestial coordinates are given in `ref[1:2]`. If undefined (NaN) the initialization routine, `set`, will set this to a projection-specific default.

See also:

`astropy.wcs.Wcsprm.phi0`

velangl

double Velocity angle.

The angle in degrees that should be used to decompose an observed velocity into radial and transverse components.

An undefined value is represented by NaN.

velosys

double Relative radial velocity.

The relative radial velocity (m/s) between the observer and the selected standard of rest in the direction of the celestial reference coordinate, VELOSYSa.

An undefined value is represented by NaN.

See also:

`astropy.wcs.Wcsprm.specsys`, `astropy.wcs.Wcsprm.ssysobs`

zsource

double The redshift, ZSOURCEa, of the source.

An undefined value is represented by NaN.

Methods Documentation

bounds_check (*pix2world*, *world2pix*)

Enable/disable bounds checking.

Parameters

pix2world : bool, optional

When `True`, enable bounds checking for the pixel-to-world (p2x) transformations. Default is `True`.

world2pix : bool, optional

When `True`, enable bounds checking for the world-to-pixel (s2x) transformations. Default is `True`.

Notes

Note that by default (without calling `bounds_check`) strict bounds checking is enabled.

cdfix()

Fix erroneously omitted `CDi_ja` keywords.

Sets the diagonal element of the `CDi_ja` matrix to unity if all `CDi_ja` keywords associated with a given axis were omitted. According to Paper I, if any `CDi_ja` keywords at all are given in a FITS header then those not given default to zero. This results in a singular matrix with an intersecting row and column of zeros.

Returns

success : int

Returns 0 for success; -1 if no change required.

celfix()

Translates AIPS-convention celestial projection types, `-NCP` and `-GLS`.

Returns

success : int

Returns 0 for success; -1 if no change required.

compare(other, cmp=0)

Compare two `Wcsprm` objects for equality.

Parameters

other : `Wcsprm`

The other `Wcsprm` object to compare to.

cmp : int, optional

A bit field controlling the strictness of the comparison. When 0, (the default), all fields must be identical.

The following constants may be or'ed together to loosen the comparison.

- `WCSCOMPARE_ANCILLARY`: Ignores ancillary keywords that don't change the WCS transformation, such as `DATE-OBS` or `EQUINOX`.
- `WCSCOMPARE_TILING`: Ignore integral differences in `CRPIXja`. This is the 'tiling' condition, where two WCSes cover different regions of the same map projection and align on the same map grid.
- `WCSCOMPARE_CRPIX`: Ignore any differences at all in `CRPIXja`. The two WCSes cover different regions of the same map projection but may not align on the same grid map. Overrides `WCSCOMPARE_TILING`.

Returns

equal : bool

cylfix()

Fixes WCS keyvalues for malformed cylindrical projections.

Returns

success : int

Returns 0 for success; -1 if no change required.

datfix()

Translates the old DATE-OBS date format to year-2000 standard form (yyyy-mm-ddThh:mm:ss) and derives MJD-OBS from it if not already set.

Alternatively, if `mjdobs` is set and `dateobs` isn't, then `datfix` derives `dateobs` from it. If both are set but disagree by more than half a day then `ValueError` is raised.

Returns

success : int

Returns 0 for success; -1 if no change required.

fix(translate_units='', naxis=0)

Applies all of the corrections handled separately by `datfix`, `unitfix`, `celfix`, `spcfix`, `cylfix` and `cdfix`.

Parameters

translate_units : str, optional

Specify which potentially unsafe translations of non-standard unit strings to perform. By default, performs all.

Although "S" is commonly used to represent seconds, its translation to "s" is potentially unsafe since the standard recognizes "S" formally as Siemens, however rarely that may be used. The same applies to "H" for hours (Henry), and "D" for days (Debye).

This string controls what to do in such cases, and is case-insensitive.

- If the string contains "s", translate "S" to "s".
- If the string contains "h", translate "H" to "h".
- If the string contains "d", translate "D" to "d".

Thus '' doesn't do any unsafe translations, whereas 'shd' does all of them.

naxis : int array[naxis], optional

Image axis lengths. If this array is set to zero or `None`, then `cylfix` will not be invoked.

Returns

status : dict

Returns a dictionary containing the following keys, each referring to a status string for each of the sub-fix functions that were called:

- `cdfix`
- `datfix`
- `unitfix`
- `celfix`
- `spcfix`
- `cylfix`

get_cdelt() → double array[naxis]

Coordinate increments (CDELTia) for each coord axis.

Returns the CDELT offsets in read-only form. Unlike the `cdelt` property, this works even when the header specifies the linear transformation matrix in one of the alternative `CDi_ja` or `CROTAia` forms.

This is useful when you want access to the linear transformation matrix, but don't care how it was specified in the header.

`get_pc()` → double array[naxis][naxis]

Returns the PC matrix in read-only form. Unlike the `pc` property, this works even when the header specifies the linear transformation matrix in one of the alternative `CDi_ja` or `CROTAia` forms. This is useful when you want access to the linear transformation matrix, but don't care how it was specified in the header.

`get_ps()` → list of tuples

Returns `PSi_ma` keywords for each i and m .

Returns

`ps`: list of tuples

Returned as a list of tuples of the form $(i, m, value)$:

- i : int. Axis number, as in `PSi_ma`, (i.e. 1-relative)
- m : int. Parameter number, as in `PSi_ma`, (i.e. 0-relative)
- $value$: string. Parameter value.

See also:

`astropy.wcs.Wcsprm.set_ps`

Set `PSi_ma` values

`get_pv()` → list of tuples

Returns `PVi_ma` keywords for each i and m .

Returns

Returned as a list of tuples of the form $(i, m, value)$:

- i : int. Axis number, as in `PVi_ma`, (i.e. 1-relative)
- m : int. Parameter number, as in `PVi_ma`, (i.e. 0-relative)
- $value$: string. Parameter value.

See also:

`astropy.wcs.Wcsprm.set_pv`

Set `PVi_ma` values

Notes

Note that, if they were not given, `set` resets the entries for `PVi_1a`, `PVi_2a`, `PVi_3a`, and `PVi_4a` for longitude axis i to match `(phi_0, theta_0)`, the native longitude and latitude of the reference point given by `LONPOLEa` and `LATPOLEa`.

`has_cd()` → bool

Returns `True` if `CDi_ja` is present.

`CDi_ja` is an alternate specification of the linear transformation matrix, maintained for historical compatibility.

Matrix elements in the IRAF convention are equivalent to the product `CDi_ja = CDELTia * PCi_ja`, but the defaults differ from that of the `PCi_ja` matrix. If one or more `CDi_ja` keywords are present then all unspecified `CDi_ja` default to zero. If no `CDi_ja` (or `CROTAia`) keywords are

present, then the header is assumed to be in `PCi_ja` form whether or not any `PCi_ja` keywords are present since this results in an interpretation of `CDELTA` consistent with the original FITS specification.

While `CDi_ja` may not formally co-exist with `PCi_ja`, it may co-exist with `CDELTA` and `CROTAia` which are to be ignored.

See also:

`astropy.wcs.Wcsprm.cd`

Get the raw `CDi_ja` values.

`has_cdi_ja()` → bool

Alias for `has_cd`. Maintained for backward compatibility.

`has_crota()` → bool

Returns `True` if `CROTAia` is present.

`CROTAia` is an alternate specification of the linear transformation matrix, maintained for historical compatibility.

In the AIPS convention, `CROTAia` may only be associated with the latitude axis of a celestial axis pair. It specifies a rotation in the image plane that is applied *after* the `CDELTA`; any other `CROTAia` keywords are ignored.

`CROTAia` may not formally co-exist with `PCi_ja`. `CROTAia` and `CDELTA` may formally co-exist with `CDi_ja` but if so are to be ignored.

See also:

`astropy.wcs.Wcsprm.crota`

Get the raw `CROTAia` values

`has_crotaia()` → bool

Alias for `has_crota`. Maintained for backward compatibility.

`has_pc()` → bool

Returns `True` if `PCi_ja` is present. `PCi_ja` is the recommended way to specify the linear transformation matrix.

See also:

`astropy.wcs.Wcsprm.pc`

Get the raw `PCi_ja` values

`has_pci_ja()` → bool

Alias for `has_pc`. Maintained for backward compatibility.

`is_unity()` → bool

Returns `True` if the linear transformation matrix (`cd`) is unity.

mix (*mixpix*, *mixcel*, *vspan*, *vstep*, *viter*, *world*, *pixcrd*, *origin*)

Given either the celestial longitude or latitude plus an element of the pixel coordinate, solves for the remaining elements by iterating on the unknown celestial coordinate element using `s2p`.

Parameters

mixpix : int

Which element on the pixel coordinate is given.

mixcel : int

Which element of the celestial coordinate is given. If *mixcel* = 1, celestial longitude is given in `world[self.lng]`, latitude returned in `world[self.lat]`. If *mixcel* = 2, celestial latitude is given in `world[self.lat]`, longitude returned in `world[self.lng]`.

vspan : pair of floats

Solution interval for the celestial coordinate, in degrees. The ordering of the two limits is irrelevant. Longitude ranges may be specified with any convenient normalization, for example $(-120, +120)$ is the same as $(240, 480)$, except that the solution will be returned with the same normalization, i.e. lie within the interval specified.

vstep : float

Step size for solution search, in degrees. If 0, a sensible, although perhaps non-optimal default will be used.

viter : int

If a solution is not found then the step size will be halved and the search recommenced. *viter* controls how many times the step size is halved. The allowed range is 5 - 10.

world : double array[naxis]

World coordinate elements. `world[self.lng]` and `world[self.lat]` are the celestial longitude and latitude, in degrees. Which is given and which returned depends on the value of *mixcel*. All other elements are given. The results will be written to this array in-place.

pixcrd : double array[naxis].

Pixel coordinates. The element indicated by *mixmap* is given and the remaining elements will be written in-place.

origin : int

Specifies the origin of pixel values. The Fortran and FITS standards use an origin of 1. Numpy and C use array indexing with origin at 0.

Returns

result : dict

Returns a dictionary with the following keys:

- *phi* (double array[naxis])
- *theta* (double array[naxis])
 - Longitude and latitude in the native coordinate system of the projection, in degrees.
- *imgcrd* (double array[naxis])
 - Image coordinate elements. `imgcrd[self.lng]` and `imgcrd[self.lat]` are the projected *x*- and *y*-coordinates, in decimal degrees.
- *world* (double array[naxis])
 - Another reference to the *world* argument passed in.

Raises

MemoryError

Memory allocation failed.

SingularMatrixError

Linear transformation matrix is singular.

InconsistentAxisTypesError

Inconsistent or unrecognized coordinate axis types.

ValueError

Invalid parameter value.

InvalidTransformError

Invalid coordinate transformation parameters.

InvalidTransformError

Ill-conditioned coordinate transformation parameters.

InvalidCoordinateError

Invalid world coordinate.

NoSolutionError

No solution found in the specified interval.

See also:

`astropy.wcs.Wcsprm.lat`, `astropy.wcs.Wcsprm.lng`

Notes

Initially, the specified solution interval is checked to see if it's a "crossing" interval. If it isn't, a search is made for a crossing solution by iterating on the unknown celestial coordinate starting at the upper limit of the solution interval and decrementing by the specified step size. A crossing is indicated if the trial value of the pixel coordinate steps through the value specified. If a crossing interval is found then the solution is determined by a modified form of "regula falsi" division of the crossing interval. If no crossing interval was found within the specified solution interval then a search is made for a "non-crossing" solution as may arise from a point of tangency. The process is complicated by having to make allowance for the discontinuities that occur in all map projections.

Once one solution has been determined others may be found by subsequent invocations of `mix` with suitably restricted solution intervals.

Note the circumstance that arises when the solution point lies at a native pole of a projection in which the pole is represented as a finite curve, for example the zenithals and conics. In such cases two or more valid solutions may exist but `mix` only ever returns one.

Because of its generality, `mix` is very compute-intensive. For compute-limited applications, more efficient special-case solvers could be written for simple projections, for example non-oblique cylindrical projections.

p2s (*pixcrd*, *origin*)

Converts pixel to world coordinates.

Parameters

pixcrd : double array[ncoord][nelem]

Array of pixel coordinates.

origin : int

Specifies the origin of pixel values. The Fortran and FITS standards use an origin of 1. Numpy and C use array indexing with origin at 0.

Returns**result** : dict

Returns a dictionary with the following keys:

- *imgcrd*: double array[ncoord][nelem]
 –Array of intermediate world coordinates. For celestial axes, `imgcrd[][self.lng]` and `imgcrd[][self.lat]` are the projected *x*-, and *y*-coordinates, in pseudo degrees. For spectral axes, `imgcrd[][self.spec]` is the intermediate spectral coordinate, in SI units.
- *phi*: double array[ncoord]
- *theta*: double array[ncoord]
 –Longitude and latitude in the native coordinate system of the projection, in degrees.
- *world*: double array[ncoord][nelem]
 –Array of world coordinates. For celestial axes, `world[][self.lng]` and `world[][self.lat]` are the celestial longitude and latitude, in degrees. For spectral axes, `world[][self.spec]` is the intermediate spectral coordinate, in SI units.
- *stat*: int array[ncoord]
 –Status return value for each coordinate. 0 for success, 1+ for invalid pixel coordinate.

Raises**MemoryError**

Memory allocation failed.

SingularMatrixError

Linear transformation matrix is singular.

InconsistentAxisTypesError

Inconsistent or unrecognized coordinate axis types.

ValueError

Invalid parameter value.

ValueError*x*- and *y*-coordinate arrays are not the same size.**InvalidTransformError**

Invalid coordinate transformation parameters.

InvalidTransformError

Ill-conditioned coordinate transformation parameters.

See also:`astropy.wcs.Wcsprm.lat`, `astropy.wcs.Wcsprm.lng`**print_contents()**Print the contents of the `Wcsprm` object to stdout. Probably only useful for debugging purposes, and may be removed in the future.

To get a string of the contents, use `repr`.

s2p (*world, origin*)

Transforms world coordinates to pixel coordinates.

Parameters

world : double array[ncoord][nelem]

Array of world coordinates, in decimal degrees.

origin : int

Specifies the origin of pixel values. The Fortran and FITS standards use an origin of 1. Numpy and C use array indexing with origin at 0.

Returns

result : dict

Returns a dictionary with the following keys:

•*phi*: double array[ncoord]

•*theta*: double array[ncoord]

–Longitude and latitude in the native coordinate system of the projection, in degrees.

•*imgcrd*: double array[ncoord][nelem]

–Array of intermediate world coordinates. For celestial axes, `imgcrd[][self.lng]` and `imgcrd[][self.lat]` are the projected *x*-, and *y*-coordinates, in pseudo “degrees”. For quadcube projections with a `CUBEFACE` axis, the face number is also returned in `imgcrd[][self.cubeface]`. For spectral axes, `imgcrd[][self.spec]` is the intermediate spectral coordinate, in SI units.

•*pixcrd*: double array[ncoord][nelem]

–Array of pixel coordinates. Pixel coordinates are zero-based.

•*stat*: int array[ncoord]

–Status return value for each coordinate. 0 for success, 1+ for invalid pixel coordinate.

Raises

MemoryError

Memory allocation failed.

SingularMatrixError

Linear transformation matrix is singular.

InconsistentAxisTypesError

Inconsistent or unrecognized coordinate axis types.

ValueError

Invalid parameter value.

InvalidTransformError

Invalid coordinate transformation parameters.

InvalidTransformError

Ill-conditioned coordinate transformation parameters.

See also:

`astropy.wcs.Wcsprm.lat`, `astropy.wcs.Wcsprm.lng`

set ()

Sets up a WCS object for use according to information supplied within it.

Note that this routine need not be called directly; it will be invoked by `p2s` and `s2p` if necessary.

Some attributes that are based on other attributes (such as `lattyp` on `ctype`) may not be correct until after `set` is called.

`set` strips off trailing blanks in all string members.

`set` recognizes the NCP projection and converts it to the equivalent SIN projection and it also recognizes GLS as a synonym for SFL. It does alias translation for the AIPS spectral types (FREQ-LSR, FELO-HEL, etc.) but without changing the input header keywords.

Raises**MemoryError**

Memory allocation failed.

SingularMatrixError

Linear transformation matrix is singular.

InconsistentAxisTypesError

Inconsistent or unrecognized coordinate axis types.

ValueError

Invalid parameter value.

InvalidTransformError

Invalid coordinate transformation parameters.

InvalidTransformError

Ill-conditioned coordinate transformation parameters.

set_ps (*ps*)

Sets `PSi_ma` keywords for each *i* and *m*.

Parameters

ps : sequence of tuples

The input must be a sequence of tuples of the form (*i*, *m*, *value*):

- *i*: int. Axis number, as in `PSi_ma`, (i.e. 1-relative)
- *m*: int. Parameter number, as in `PSi_ma`, (i.e. 0-relative)
- *value*: string. Parameter value.

See also:

`astropy.wcs.Wcsprm.get_ps`

set_pv (*pv*)

Sets `PVi_ma` keywords for each *i* and *m*.

Parameters

pv : list of tuples

The input must be a sequence of tuples of the form (*i*, *m*, *value*):

- i*: int. Axis number, as in `PVi_ma`, (i.e. 1-relative)
- m*: int. Parameter number, as in `PVi_ma`, (i.e. 0-relative)
- value*: float. Parameter value.

See also:

`astropy.wcs.Wcsprm.get_pv`

spcfix() → int

Translates AIPS-convention spectral coordinate types. {FREQ, VELO, FELO}-{OBS, HEL, LSR} (e.g. FREQ-LSR, VELO-OBS, FELO-HEL)

Returns

success : int

Returns 0 for success; -1 if no change required.

sptr(*ctype*, *i=-1*)

Translates the spectral axis in a WCS object.

For example, a FREQ axis may be translated into ZOPT-F2W and vice versa.

Parameters

ctype : str

Required spectral `CTYPEi`, maximum of 8 characters. The first four characters are required to be given and are never modified. The remaining four, the algorithm code, are completely determined by, and must be consistent with, the first four characters. Wildcarding may be used, i.e. if the final three characters are specified as "???", or if just the eighth character is specified as "?", the correct algorithm code will be substituted and returned.

i : int

Index of the spectral axis (0-relative). If `i < 0` (or not provided), it will be set to the first spectral axis identified from the `CTYPE` keyvalues in the FITS header.

Raises**MemoryError**

Memory allocation failed.

SingularMatrixError

Linear transformation matrix is singular.

InconsistentAxisTypesError

Inconsistent or unrecognized coordinate axis types.

ValueError

Invalid parameter value.

InvalidTransformError

Invalid coordinate transformation parameters.

InvalidTransformError

Ill-conditioned coordinate transformation parameters.

InvalidSubimageSpecificationError

Invalid subimage specification (no spectral axis).

sub (*axes*)

Extracts the coordinate description for a subimage from a `WCS` object.

The world coordinate system of the subimage must be separable in the sense that the world coordinates at any point in the subimage must depend only on the pixel coordinates of the axes extracted. In practice, this means that the `PCi_ja` matrix of the original image must not contain non-zero off-diagonal terms that associate any of the subimage axes with any of the non-subimage axes.

`sub` can also add axes to a `wcsprm` object. The new axes will be created using the defaults set by the `Wcsprm` constructor which produce a simple, unnamed, linear axis with world coordinates equal to the pixel coordinate. These default values can be changed before invoking `set`.

Parameters

axes : int or a sequence.

- If an int, include the first N axes in their original order.
- If a sequence, may contain a combination of image axis numbers (1-relative) or special axis identifiers (see below). Order is significant; `axes[0]` is the axis number of the input image that corresponds to the first axis in the subimage, etc. Use an axis number of 0 to create a new axis using the defaults.
- If 0, [] or None, do a deep copy.

Coordinate axes types may be specified using either strings or special integer constants. The available types are:

- `'longitude'` / `WCSSUB_LONGITUDE`: Celestial longitude
- `'latitude'` / `WCSSUB_LATITUDE`: Celestial latitude
- `'cubeface'` / `WCSSUB_CUBEFACE`: `Quadcube CUBEFACE` axis
- `'spectral'` / `WCSSUB_SPECTRAL`: Spectral axis
- `'stokes'` / `WCSSUB_STOKES`: Stokes axis
- `'celestial'` / `WCSSUB_CELESTIAL`: An alias for the combination of `'longitude'`, `'latitude'` and `'cubeface'`.

Returns

new_wcs : `WCS` object

Raises**MemoryError**

Memory allocation failed.

InvalidSubimageSpecificationError

Invalid subimage specification (no spectral axis).

NonseparableSubimageCoordinateSystem

Non-separable subimage coordinate system.

Notes

Combinations of subimage axes of particular types may be extracted in the same order as they occur in the input image by combining the integer constants with the ‘binary or’ (`|`) operator. For example:

```
wcs.sub([WCSSUB_LONGITUDE | WCSSUB_LATITUDE | WCSSUB_SPECTRAL])
```

would extract the longitude, latitude, and spectral axes in the same order as the input image. If one of each were present, the resulting object would have three dimensions.

For convenience, `WCSSUB_CELESTIAL` is defined as the combination `WCSSUB_LONGITUDE | WCSSUB_LATITUDE | WCSSUB_CUBEFACE`.

The codes may also be negated to extract all but the types specified, for example:

```
wcs.sub([
    WCSSUB_LONGITUDE,
    WCSSUB_LATITUDE,
    WCSSUB_CUBEFACE,
    -(WCSSUB_SPECTRAL | WCSSUB_STOKES)])
```

The last of these specifies all axis types other than spectral or Stokes. Extraction is done in the order specified by `axes`, i.e. a longitude axis (if present) would be extracted first (via `axes[0]`) and not subsequently (via `axes[3]`). Likewise for the latitude and cubeface axes in this example.

The number of dimensions in the returned object may be less than or greater than the length of `axes`. However, it will never exceed the number of axes in the input image.

to_header (*relax=False*)

`to_header` translates a WCS object into a FITS header.

The details of the header depends on context:

- If the `colnum` member is non-zero then a binary table image array header will be produced.
- Otherwise, if the `colax` member is set non-zero then a pixel list header will be produced.
- Otherwise, a primary image or image extension header will be produced.

The output header will almost certainly differ from the input in a number of respects:

- 1.The output header only contains WCS-related keywords. In particular, it does not contain syntactically-required keywords such as `SIMPLE`, `NAXIS`, `BITPIX`, or `END`.
- 2.Deprecated (e.g. `CROTAN`) or non-standard usage will be translated to standard (this is partially dependent on whether `fix` was applied).
- 3.Quantities will be converted to the units used internally, basically SI with the addition of degrees.
- 4.Floating-point quantities may be given to a different decimal precision.
- 5.Elements of the `PCi,j` matrix will be written if and only if they differ from the unit matrix. Thus, if the matrix is unity then no elements will be written.
- 6.Additional keywords such as `WCSAXES`, `CUNITia`, `LONPOLEa` and `LATPOLEa` may appear.
- 7.The original keycomments will be lost, although `to_header` tries hard to write meaningful comments.
- 8.Keyword order may be changed.

Keywords can be translated between the image array, binary table, and pixel lists forms by manipulating the `colnum` or `colax` members of the `WCS` object.

Parameters

relax : bool or int

Degree of permissiveness:

- False**: Recognize only FITS keywords defined by the published WCS standard.
- True**: Admit all recognized informal extensions of the WCS standard.

- `int`: a bit field selecting specific extensions to write. See *Header-writing relaxation constants* for details.

Returns**header** : str

Raw FITS header as a string.

unitfix (*translate_units=''*)

Translates non-standard CUNITia keyvalues.

For example, DEG -> deg, also stripping off unnecessary whitespace.

Parameters**translate_units** : str, optional

Do potentially unsafe translations of non-standard unit strings.

Although "S" is commonly used to represent seconds, its recognizes "S" formally as Siemens, however rarely that may be translation to "s" is potentially unsafe since the standard used. The same applies to "H" for hours (Henry), and "D" for days (Debye).

This string controls what to do in such cases, and is case-insensitive.

- If the string contains "s", translate "S" to "s".
- If the string contains "h", translate "H" to "h".
- If the string contains "d", translate "D" to "d".

Thus '' doesn't do any unsafe translations, whereas 'shd' does all of them.

Returns**success** : int

Returns 0 for success; -1 if no change required.

Class Inheritance Diagram

11.8 Acknowledgments and Licenses

wcslib is licenced under the [GNU Lesser General Public License](#).

MODELS AND FITTING (ASTROPY . MODELING)

12.1 Introduction

`astropy.modeling` provides a framework for representing models and performing model evaluation and fitting. It currently supports 1-D and 2-D models and fitting with parameter constraints.

It is *designed* to be easily extensible and flexible. Models do not reference fitting algorithms explicitly and new fitting algorithms may be added without changing the existing models (though not all models can be used with all fitting algorithms due to constraints such as model linearity).

The goal is to eventually provide a rich toolset of models and fitters such that most users will not need to define new model classes, nor special purpose fitting routines (while making it reasonably easy to do when necessary).

Warning: `astropy.modeling` is currently a work-in-progress, and thus it is likely there will be significant API changes in later versions of Astropy. If you have specific ideas for how it might be improved, feel free to let us know on the [astropy-dev mailing list](mailto:astropy-dev@python.org) or at <http://feedback.astropy.org>

12.2 Getting started

The examples here use the predefined models and assume the following modules have been imported:

```
>>> import numpy as np
>>> from astropy.modeling import models, fitting
```

12.2.1 Using Models

The `astropy.modeling` package defines a number of models that are collected under a single namespace as `astropy.modeling.models`. Models behave like parametrized functions:

```
>>> from astropy.modeling import models
>>> g = models.Gaussian1D(amplitude=1.2, mean=0.9, stddev=0.5)
>>> print(g)
Model: Gaussian1D
Inputs: 1
Outputs: 1
Model set size: 1
Parameters:
  amplitude mean stddev
  -----
      1.2    0.9    0.5
```

Model parameters can be accessed as attributes:

```
>>> g.amplitude
Parameter('amplitude', value=1.2)
>>> g.mean
Parameter('mean', value=0.9)
>>> g.stddev
Parameter('stddev', value=0.5)
```

and can also be updated via those attributes:

```
>>> g.amplitude = 0.8
>>> g.amplitude
Parameter('amplitude', value=0.8)
```

Models can be evaluated by calling them as functions:

```
>>> g(0.1)
0.22242984036255528
>>> g(np.linspace(0.5, 1.5, 7))
array([ 0.58091923,  0.71746405,  0.7929204 ,  0.78415894,  0.69394278,
        0.54952605,  0.3894018 ])
```

As the above example demonstrates, in general most models evaluate array-like inputs according to the standard [Numpy broadcasting rules](#) for arrays.

Models can therefore already be useful to evaluate common functions, independently of the fitting features of the package.

12.2.2 Simple 1-D model fitting

In this section, we look at a simple example of fitting a Gaussian to a simulated dataset. We use the `Gaussian1D` and `Trapezoid1D` models and the `LevMarLSQFitter` fitter to fit the data:

```
import numpy as np
from astropy.modeling import models, fitting

# Generate fake data
np.random.seed(0)
x = np.linspace(-5., 5., 200)
y = 3 * np.exp(-0.5 * (x - 1.3)**2 / 0.8**2)
y += np.random.normal(0., 0.2, x.shape)

# Fit the data using a box model
t_init = models.Trapezoid1D(amplitude=1., x_0=0., width=1., slope=0.5)
fit_t = fitting.LevMarLSQFitter()
t = fit_t(t_init, x, y)

# Fit the data using a Gaussian
g_init = models.Gaussian1D(amplitude=1., mean=0, stddev=1.)
fit_g = fitting.LevMarLSQFitter()
g = fit_g(g_init, x, y)

# Plot the data with the best-fit model
plt.figure(figsize=(8,5))
plt.plot(x, y, 'ko')
plt.plot(x, t(x), 'b-', lw=2, label='Trapezoid')
plt.plot(x, g(x), 'r-', lw=2, label='Gaussian')
plt.xlabel('Position')
```

```
plt.ylabel('Flux')
plt.legend(loc=2)
```

As shown above, once instantiated, the fitter class can be used as a function that takes the initial model (`t_init` or `g_init`) and the data values (`x` and `y`), and returns a fitted model (`t` or `g`).

12.2.3 Simple 2-D model fitting

Similarly to the 1-D example, we can create a simulated 2-D data dataset, and fit a polynomial model to it. This could be used for example to fit the background in an image.

```
import numpy as np
from astropy.modeling import models, fitting

# Generate fake data
np.random.seed(0)
y, x = np.mgrid[:128, :128]
z = 2. * x ** 2 - 0.5 * x ** 2 + 1.5 * x * y - 1.
z += np.random.normal(0., 0.1, z.shape) * 50000.

# Fit the data using astropy.modeling
p_init = models.Polynomial2D(degree=2)
fit_p = fitting.LevMarLSQFitter()
p = fit_p(p_init, x, y, z)

# Plot the data with the best-fit model
plt.figure(figsize=(8, 2.5))
plt.subplot(1, 3, 1)
plt.imshow(z, interpolation='nearest', vmin=-1e4, vmax=5e4)
plt.title("Data")
plt.subplot(1, 3, 2)
plt.imshow(p(x, y), interpolation='nearest', vmin=-1e4, vmax=5e4)
plt.title("Model")
plt.subplot(1, 3, 3)
plt.imshow(z - p(x, y), interpolation='nearest', vmin=-1e4, vmax=5e4)
plt.title("Residual")
```

A list of models is provided in the [Reference/API](#) section. The fitting framework includes many useful features that are not demonstrated here, such as weighting of datapoints, fixing or linking parameters, and placing lower or upper limits on parameters. For more information on these, take a look at the [Fitting Models to Data](#) documentation.

12.2.4 Model sets

In some cases it is necessary to describe many models of the same type but with different parameter values. This could be done simply by instantiating as many instances of a `Model` as are needed. But that can be inefficient for a large number of models. To that end, all model classes in `astropy.modeling` can also be used to represent a model *set* which is a collection of models of the same type, but with different values for their parameters.

To instantiate a model set, use argument `n_models=N` where `N` is the number of models in the set when constructing the model. The value of each parameter must be a list or array of length `N`, such that each item in the array corresponds to one model in the set:

```
>>> g = models.Gaussian1D(amplitude=[1, 2], mean=[0, 0],
...                       stddev=[0.1, 0.2], n_models=2)
>>> print(g)
Model: Gaussian1D
```

```
Inputs: 1
Outputs: 1
Model set size: 2
Parameters:
  amplitude mean stddev
-----
      1.0  0.0   0.1
      2.0  0.0   0.2
```

This is equivalent to two Gaussians with the parameters `amplitude=1`, `mean=0`, `stddev=0.1` and `amplitude=2`, `mean=0`, `stddev=0.2` respectively. When printing the model the parameter values are displayed as a table, with each row corresponding to a single model in the set.

The number of models in a model set can be determined using the `len` builtin:

```
>>> len(g)
2
```

Single models have a length of 1, and are not considered a model set as such.

When evaluating a model set, by default the input must be the same length as the number of models, with one input per model:

```
>>> g([0, 0.1])
array([ 1.          ,  1.76499381])
```

The result is an array with one result per model in the set. It is also possible to broadcast a single value to all models in the set:

```
>>> g(0)
array([ 1.,  2.])
```

Model sets are used primarily for fitting, allowing a large number of models of the same type to be fitted simultaneously (and independently from each other) to some large set of inputs. For example, fitting a polynomial to the time response of each pixel in a data cube. This can greatly speed up the fitting process, especially for linear models.

12.3 Using `astropy.modeling`

12.3.1 Parameters

Most models in this package are “parametric” in the sense that each subclass of `Model` represents an entire family of models, each member of which is distinguished by a fixed set of parameters that fit that model to some some dependent and independent variable(s) (also referred to throughout the the package as the outputs and inputs of the model).

Parameters are used in three different contexts within this package: Basic evaluation of models, fitting models to data, and providing information about individual models to users (including documentation).

Most subclasses of `Model`—specifically those implementing a specific physical or statistical model, have a fixed set of parameters that can be specified for instances of that model. There are a few classes of models (in particular polynomials) in which the number of parameters depends on some other property of the model (the degree in the case of polynomials).

Models maintain a list of parameter names, `param_names`. Single parameters are instances of `Parameter` which provide a proxy for the actual parameter values. Simple mathematical operations can be performed with them, but they also contain additional attributes specific to model parameters, such as any constraints on their values and documentation.

Parameter values may be scalars *or* array values. Some parameters are required by their very nature to be arrays (such as the transformation matrix for an `AffineTransformation2D`). In most other cases, however, array-valued parameters have no meaning specific to the model, and are simply combined with input arrays during model evaluation according to the standard [Numpy broadcasting rules](#).

Parameter examples

- Model classes can be introspected directly to find out what parameters they accept:

```
>>> from astropy.modeling import models
>>> models.Gaussian1D.param_names
['amplitude', 'mean', 'stddev']
```

The order of the items in the `param_names` list is relevant—this is the same order in which values for those parameters should be passed in when constructing an instance of that model:

```
>>> g = models.Gaussian1D(1.0, 0.0, 0.1)
>>> g
<Gaussian1D(amplitude=1.0, mean=0.0, stddev=0.1...)>
```

However, parameters may also be given as keyword arguments (in any order):

```
>>> g = models.Gaussian1D(mean=0.0, amplitude=2.0, stddev=0.2)
>>> g
<Gaussian1D(amplitude=2.0, mean=0.0, stddev=0.2...)>
```

So all that really matters is knowing the names (and meanings) of the parameters that each model accepts. More information about an individual model can also be obtained using the `help` built-in:

```
>>> help(models.Gaussian1D)
```

- Some types of models can have different numbers of parameters depending on other properties of the model. In particular, the parameters of polynomial models are their coefficients, the number of which depends on the polynomial's degree:

```
>>> p1 = models.Polynomial1D(degree=3, c0=1.0, c1=0.0, c2=2.0, c3=3.0)
>>> p1.param_names
['c0', 'c1', 'c2', 'c3']
>>> p1
<Polynomial1D(3, c0=1.0, c1=0.0, c2=2.0, c3=3.0)>
```

For the basic `Polynomial1D` class the parameters are named `c0` through `cN` where `N` is the degree of the polynomial. The above example represents the polynomial $3x^3 + 2x^2 + 1$.

- Some models also have default values for one or more of their parameters. For polynomial models, for example, the default value of all coefficients is zero—this allows a polynomial instance to be created without specifying any of the coefficients initially:

```
>>> p2 = models.Polynomial1D(degree=4)
>>> p2
<Polynomial1D(4, c0=0.0, c1=0.0, c2=0.0, c3=0.0, c4=0.0)>
```

- Parameters can be set/updated by accessing attributes on the model of the same names as the parameters:

```
>>> p2.c4 = 1
>>> p2.c2 = 3.5
>>> p2.c0 = 2.0
>>> p2
<Polynomial1D(4, c0=2.0, c1=0.0, c2=3.5, c3=0.0, c4=1.0)>
```

This example now represents the polynomial $x^4 + 3.5x^2 + 2$.

- It is possible to set the coefficients of a polynomial by passing the parameters in a dictionary, since all parameters can be provided as keyword arguments:

```
>>> ch2 = models.Chebyshev2D(x_degree=2, y_degree=3)
>>> coeffs = dict((name, [idx, idx + 10])
...                 for idx, name in enumerate(ch2.param_names))
>>> ch2 = models.Chebyshev2D(x_degree=2, y_degree=3, n_models=2,
...                           **coeffs)
...
>>> ch2.param_sets
array([[ 0., 10.],
       [ 1., 11.],
       [ 2., 12.],
       [ 3., 13.],
       [ 4., 14.],
       [ 5., 15.],
       [ 6., 16.],
       [ 7., 17.],
       [ 8., 18.],
       [ 9., 19.],
       [10., 20.],
       [11., 21.]])
```

- Or directly, using keyword arguments:

```
>>> ch2 = models.Chebyshev2D(x_degree=2, y_degree=3,
...                           c0_0=[0, 10], c0_1=[3, 13],
...                           c0_2=[6, 16], c0_3=[9, 19],
...                           c1_0=[1, 11], c1_1=[4, 14],
...                           c1_2=[7, 17], c1_3=[10, 20],
...                           c2_0=[2, 12], c2_1=[5, 15],
...                           c2_2=[8, 18], c2_3=[11, 21])
```

- Individual parameters values may be arrays of different sizes and shapes:

```
>>> p3 = models.Polynomial1D(degree=2, c0=1.0, c1=[2.0, 3.0],
...                          c2=[[4.0, 5.0], [6.0, 7.0], [8.0, 9.0]])
...
>>> p3(2.0)
array([[ 21., 27.],
       [ 29., 35.],
       [ 37., 43.]])
```

This is equivalent to evaluating the Numpy expression:

```
>>> import numpy as np
>>> c2 = np.array([[4.0, 5.0],
...                [6.0, 7.0],
...                [8.0, 9.0]])
>>> c1 = np.array([2.0, 3.0])
>>> c2 * 2.0**2 + c1 * 2.0 + 1.0
array([[ 21., 27.],
       [ 29., 35.],
       [ 37., 43.]])
```

Note that in most cases, when using array-valued parameters, the parameters must obey the standard broadcasting rules for Numpy arrays with respect to each other:

```
>>> models.Polynomial1D(degree=2, c0=1.0, c1=[2.0, 3.0],
...                    c2=[4.0, 5.0, 6.0])
```

```

Traceback (most recent call last):
...
InputParameterError: Parameter u'c1' of shape (2,) cannot be broadcast
with parameter u'c2' of shape (3,). All parameter arrays must have
shapes that are mutually compatible according to the broadcasting rules.
Traceback (most recent call last):
...
InputParameterError: Parameter u'c1' of shape (2,) cannot be broadcast

```

12.3.2 Instantiating and Evaluating Models

The base class of all models is `Model`, however fittable models should subclass `FittableModel`. Fittable models can be linear or nonlinear in a regression analysis sense.

In general models are instantiated by providing the parameter values that define that instance of the model to the constructor, as demonstrated in the section on *Parameters*.

Additionally, a `Model` instance may represent a single model with one set of parameters, or a model *set* consisting of a set of parameters each representing a different parameterization of the same parametric model. For example one may instantiate a single Gaussian model with one mean, standard deviation, and amplitude. Or one may create a set of N Gaussians, each one of which would be fitted to, for example, a different plane in an image cube.

Regardless of whether using a single model, or a model set, parameter values may be scalar values, or arrays of any size and shape, so long as they are compatible according to the standard [Numpy broadcasting rules](#). For example, a model may be instantiated with all scalar parameters:

```

>>> from astropy.modeling.models import Gaussian1D
>>> g = Gaussian1D(amplitude=1, mean=0, stddev=1)
>>> g
<Gaussian1D(amplitude=1.0, mean=0.0, stddev=1.0)>

```

Or it may use all array parameters. For example if all parameters are 2x2 arrays the model is computed element-wise using all elements in the arrays:

```

>>> g = Gaussian1D(amplitude=[[1, 2], [3, 4]], mean=[[0, 1], [1, 0]],
...               stddev=[[0.1, 0.2], [0.3, 0.4]])
>>> g
<Gaussian1D(amplitude=[[ 1., 2.], [ 3., 4.]], mean=[[ 0., 1.], [ 1., 0.]],
stddev=[[ 0.1, 0.2], [ 0.3, 0.4]])>
>>> g(0)
array([[ 1.00000000e+00,  7.45330634e-06],
       [ 1.15977604e-02,  4.00000000e+00]])

```

Or it may even use a mix of scalar values and arrays of different sizes and dimensions so long as they are compatible:

```

>>> g = Gaussian1D(amplitude=[[1, 2], [3, 4]], mean=0.1, stddev=[0.1, 0.2])
>>> g(0)
array([[ 0.60653066,  1.76499381],
       [ 1.81959198,  3.52998761]])

```

In this case, four values are computed—one using each element of the amplitude array. Each model uses a mean of 0.1, and a standard deviation of 0.1 is used with the amplitudes of 1 and 3, and 0.2 is used with amplitudes 2 and 4.

If any of the parameters have incompatible values this will result in an error:

```

>>> g = Gaussian1D(amplitude=1, mean=[1, 2], stddev=[1, 2, 3])
Traceback (most recent call last):
...

```

```
InputParameterError: Parameter 'mean' of shape (2,) cannot be broadcast
with parameter 'stddev' of shape (3,). All parameter arrays must have
shapes that are mutually compatible according to the broadcasting rules.
Traceback (most recent call last):
...
InputParameterError: Parameter 'mean' of shape (2,) cannot be broadcast
```

Model Sets

By default, `Model` instances represent a single model. There are two ways, when instantiating a `Model` instance, to create a model set instead. The first is to specify the `n_models` argument when instantiating the model:

```
>>> g = Gaussian1D(amplitude=[1, 2], mean=[0, 0], stddev=[0.1, 0.2],
...                n_models=2)
>>> g
<Gaussian1D(amplitude=[ 1., 2.], mean=[ 0., 0.], stddev=[ 0.1, 0.2],
n_models=2)>
```

When specifying some `n_models=N` this requires that the parameter values be arrays of some kind, the first *axis* of which has as length of `N`. This axis is referred to as the `model_set_axis`, and by default is the 0th axis of parameter arrays. In this case the parameters were given as 1-D arrays of length 2. The values `amplitude=1`, `mean=0`, `stddev=0.1` are the parameters for the first model in the set. The values `amplitude=2`, `mean=0`, `stddev=0.2` are the parameters defining the second model in the set.

This has different semantics from simply using array values for the parameters, in that ensures that parameter values and input values are matched up according to the `model_set_axis` before any other array broadcasting rules are applied.

For example, in the previous section we created a model with array values like:

```
>>> g = Gaussian1D(amplitude=[[1, 2], [3, 4]], mean=0.1, stddev=[0.1, 0.2])
```

If instead we treat the rows as values for two different model sets, this particular instantiation will fail, since only one value is given for `mean`:

```
>>> g = Gaussian1D(amplitude=[[1, 2], [3, 4]], mean=0.1, stddev=[0.1, 0.2],
...                n_models=2)
Traceback (most recent call last):
...
InputParameterError: All parameter values must be arrays of dimension at
least 1 for model_set_axis=0 (the value given for 'mean' is only
0-dimensional)
Traceback (most recent call last):
...
InputParameterError: All parameter values must be arrays of dimension at
```

To get around this for now, provide two values for `mean`:

```
>>> g = Gaussian1D(amplitude=[[1, 2], [3, 4]], mean=[0.1, 0.1],
...                stddev=[0.1, 0.2], n_models=2)
```

This is different from the case without `n_models=2`. It does not mean that the value of `amplitude` is a 2x2 array. Rather, it means there are *two* values for `amplitude` (one for each model in the set), each of which is 1-D array of length 2. The value for the first model is `[1, 2]`, and the value for the second model is `[3, 4]`. Likewise, scalar values are given for the `mean` and standard deviation of each model in the set.

When evaluating this model on a single input we get a different result from the single-model case:

```
>>> g(0)
array([[ 0.60653066,  1.21306132],
       [ 2.64749071,  3.52998761]])
```

Each row in this output is the output for each model in the set. The first is the value of the Gaussian with `amplitude=[1, 2]`, `mean=0.1`, `stddev=0.1`, and the second is the value of the Gaussian with `amplitude=[3, 4]`, `mean=0.1`, `stddev=0.2`.

We can also pass a different input to each model in a model set by passing in an array input:

```
>>> g([0, 1])
array([[ 6.06530660e-01,  1.21306132e+00],
       [ 1.20195892e-04,  1.60261190e-04]])
```

By default this uses the same concept of a `model_set_axis`. The first dimension of the input array is used to map inputs to corresponding models in the model set. We can use this, for example, to evaluate the model on 1-D array inputs with a different input to each model set:

```
>>> g([[0, 1], [2, 3]])
array([[ 6.06530660e-01,  5.15351422e-18],
       [ 7.57849134e-20,  8.84815213e-46]])
```

In this case the first model is evaluated on `[0, 1]`, and the second model is evaluated on `[2, 3]`. If the input has length greater than the number of models in the set then this is in error:

```
>>> g([0, 1, 2])
Traceback (most recent call last):
...
ValueError: Input argument 'x' does not have the correct dimensions in
model_set_axis=0 for a model set with n_models=2.
Traceback (most recent call last):
...
ValueError: Input argument 'x' does not have the correct dimensions in
```

And input like `[0, 1, 2]` wouldn't work anyways because it is not compatible with the array dimensions of the parameter values. However, what if we wanted to evaluate all models in the set on the input `[0, 1]`? We could do this by simply repeating:

```
>>> g([[0, 1], [0, 1]])
array([[ 6.06530660e-01,  5.15351422e-18],
       [ 2.64749071e+00,  1.60261190e-04]])
```

But there is a workaround for this use case that does not necessitate duplication. This is to include the argument `model_set_axis=False`:

```
>>> g([0, 1], model_set_axis=False)
array([[ 6.06530660e-01,  5.15351422e-18],
       [ 2.64749071e+00,  1.60261190e-04]])
```

What `model_set_axis=False` implies is that an array-like input should not be treated as though any of its dimensions map to models in a model set. And rather, the given input should be used to evaluate all the models in the model set. For scalar inputs like `g(0)`, `model_set_axis=False` is implied automatically. But for array inputs it is necessary to avoid ambiguity.

Inputs and Outputs

Models have an `n_inputs` attribute, which shows how many coordinates the model expects as an input. All models expect coordinates as separate arguments. For example a 2-D model expects `x` and `y` coordinate values to be passed separately, i.e. as two scalars or array-like values.

Models also have an attribute `n_outputs`, which shows the number of output coordinates. The `n_inputs` and `n_outputs` attributes can be used when chaining transforms by adding models in `series` or in `parallel`. Because composite models can be nested within other composite models, creating theoretically infinitely complex models, a mechanism to map input data to models is needed. In this case the input may be wrapped in a `LabeledInput` object—a dict-like object whose items are `{label: data}` pairs.

Further examples

The examples here assume this import statement was executed:

```
>>> from astropy.modeling.models import Gaussian1D, Polynomial1D
>>> import numpy as np
```

- Create a model set of two 1-D Gaussians:

```
>>> x = np.arange(1, 10, .1)
>>> g1 = Gaussian1D(amplitude=[10, 9], mean=[2, 3],
...                stddev=[0.15, .1], n_models=2)
>>> print g1
Model: Gaussian1D
Inputs: 1
Outputs: 1
Model set size: 2
Parameters:
  amplitude mean stddev
-----
      10.0   2.0   0.15
       9.0   3.0    0.1
```

Evaluate all models in the set on one set of input coordinates:

```
>>> y = g1(x, model_set_axis=False) # broadcast the array to all models
>>> print(y.shape)
(2, 90)
```

or different inputs for each model in the set:

```
>>> y = g1([x, x + 3])
>>> print(y.shape)
(2, 90)
```

- Evaluating a set of multiple polynomial models with one input data set creates multiple output data sets:

```
>>> p1 = Polynomial1D(degree=1, n_models=5)
>>> p1.c1 = [0, 1, 2, 3, 4]
>>> print p1
Model: Polynomial1D
Inputs: 1
Outputs: 1
Model set size: 5
Degree: 1
Parameters:
  c0  c1
--- ---
  0.0 0.0
  0.0 1.0
  0.0 2.0
  0.0 3.0
```

```
0.0 4.0
>>> y = p1(x, model_set_axis=False)
```

- When passed a 2-D array, the same polynomial will map each row of the array to one model in the set, one for one:

```
>>> x = np.arange(30).reshape(5, 6)
>>> y = p1(x)
>>> y
array([[ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 6.,  7.,  8.,  9., 10., 11.],
       [24., 26., 28., 30., 32., 34.],
       [54., 57., 60., 63., 66., 69.],
       [96., 100., 104., 108., 112., 116.]])
>>> print(y.shape)
(5, 6)
```

Composite model examples

Note: Composite models in Astropy are currently in the process of being reworked, but in the meantime the existing implementation is still useful.

Create and evaluate a parallel composite model:

```
>>> from astropy.modeling import SummedCompositeModel
>>> from astropy.modeling.models import Polynomial1D, Gaussian1D
>>> x = np.arange(1,10,.1)
>>> p1 = Polynomial1D(1)
>>> g1 = Gaussian1D(amplitude=10., stddev=2.1, mean=4.2)
>>> sum_of_models = SummedCompositeModel([g1, p1])
>>> y = sum_of_models(x)
```

This is equivalent to applying the two models in parallel:

```
>>> y = x + g1(x) + p1(x)
```

In more complex cases the input and output may be mapped to transformations:

```
>>> from astropy.modeling import SerialCompositeModel
>>> from astropy.modeling.models import Polynomial2D, Shift
>>> y, x = np.mgrid[:5, :5]
>>> off = Shift(-3.2)
>>> poly2 = Polynomial2D(2)
>>> serial_composite_model = SerialCompositeModel(
...     [off, poly2], inmap=[['x'], ['x', 'y']], outmap=[['x'], ['z']])
```

The above composite transform will apply an inplace shift to x, followed by a 2-D polynomial and will save the result in an array, labeled 'z'. To evaluate this model use a `LabeledInput` object:

```
>>> from astropy.modeling import LabeledInput
>>> labeled_data = LabeledInput([x, y], ['x', 'y'])
>>> result = serial_composite_model(labeled_data)
```

The output is also a `LabeledInput` object and the result is stored in label 'z':

```
>>> print(result)
{'x': array([[ -3.2,  -2.2,  -1.2,  -0.2,   0.8],
```

```
[-3.2, -2.2, -1.2, -0.2, 0.8],
[-3.2, -2.2, -1.2, -0.2, 0.8],
[-3.2, -2.2, -1.2, -0.2, 0.8],
[-3.2, -2.2, -1.2, -0.2, 0.8]])},
'y': array([[0, 0, 0, 0, 0],
            [1, 1, 1, 1, 1],
            [2, 2, 2, 2, 2],
            [3, 3, 3, 3, 3],
            [4, 4, 4, 4, 4]]),
'z': array([[ 0.,  0.,  0.,  0.,  0.],
            [ 0.,  0.,  0.,  0.,  0.],
            [ 0.,  0.,  0.,  0.,  0.],
            [ 0.,  0.,  0.,  0.,  0.],
            [ 0.,  0.,  0.,  0.,  0.]])}
```

12.3.3 Fitting Models to Data

This module provides wrappers, called Fitters, around some Numpy and Scipy fitting functions. All Fitters can be called as functions. They take an instance of `FittableModel` as input and modify its `parameters` attribute. The idea is to make this extensible and allow users to easily add other fitters.

Linear fitting is done using Numpy's `numpy.linalg.lstsq` function. There are currently two non-linear fitters which use `scipy.optimize.leastsq` and `scipy.optimize.fmin_slsqp`.

The rules for passing input to fitters are:

- Non-linear fitters currently work only with single models (not model sets).
- The linear fitter can fit a single input to multiple model sets creating multiple fitted models. This may require specifying the `model_set_axis` argument just as used when evaluating models; this may be required for the fitter to know how to broadcast the input data.

Fitting examples

- Fitting a polynomial model to multiple data sets simultaneously:

```
>>> from astropy.modeling import models, fitting
>>> import numpy as np
>>> p1 = models.Polynomial1D(3)
>>> p1.c0 = 1
>>> p1.c1 = 2
>>> print(p1)
Model: Polynomial1D
Inputs: 1
Outputs: 1
Model set size: 1
Degree: 3
Parameters:
   c0  c1  c2  c3
---  ---  ---  ---
   1.0 2.0 0.0 0.0
>>> x = np.arange(10)
>>> y = p1(x)
>>> yy = np.array([y, y])
>>> p2 = models.Polynomial1D(3, n_models=2)
>>> pfit = fitting.LinearLSQFitter()
>>> new_model = pfit(p2, x, yy)
```

```
>>> print(new_model)
Model: Polynomial1D
Inputs: 1
Outputs: 1
Model set size: 2
Degree: 3
Parameters:
      c0  c1          c2          c3
-----
1.0 2.0 -5.86673908219e-16 3.61636197841e-17
1.0 2.0 -5.86673908219e-16 3.61636197841e-17
```

Fitters support constrained fitting.

- All fitters support fixed (frozen) parameters through the `fixed` argument to models or setting the `fixed` attribute directly on a parameter.

For linear fitters, freezing a polynomial coefficient means that a polynomial without that term will be fitted to the data. For example, fixing `c0` in a polynomial model will fit a polynomial with the zero-th order term missing. However, the fixed value of the coefficient is used when evaluating the model:

```
>>> x = np.arange(1, 10, .1)
>>> p1 = models.Polynomial1D(2, c0=[1, 1], c1=[2, 2], c2=[3, 3],
...                          n_models=2)
>>> p1
<Polynomial1D(2, c0=[ 1., 1.], c1=[ 2., 2.], c2=[ 3., 3.], n_models=2)>
>>> y = p1(x, model_set_axis=False)
>>> p1.c0.fixed = True
>>> pfit = fitting.LinearLSQFitter()
>>> new_model = pfit(p1, x, y)
>>> print(new_model)
Model: Polynomial1D
Inputs: 1
Outputs: 1
Model set size: 2
Degree: 2
Parameters:
      c0  c1          c2
-----
1.0 2.38641216243 2.96827885742
1.0 2.38641216243 2.96827885742
```

- A parameter can be `tied` (linked to another parameter). This can be done in two ways:

```
>>> def tiedfunc(g1):
...     mean = 3 * g1.stddev
...     return mean
>>> g1 = models.Gaussian1D(amplitude=10., mean=3, stddev=.5,
...                        tied={'mean': tiedfunc})
```

or:

```
>>> g1 = models.Gaussian1D(amplitude=10., mean=3, stddev=.5)
>>> g1.mean.tied = tiedfunc
```

Bounded fitting is supported through the `bounds` arguments to models or by setting `min` and `max` attributes on a parameter. Bounds for the `LevMarLSQFitter` are always exactly satisfied—if the value of the parameter is outside the fitting interval, it will be reset to the value at the bounds. The `SLSQPLSQFitter` handles bounds internally.

- Different fitters support different types of constraints:

```
>>> fitting.LinearLSQFitter.supported_constraints
['fixed']
>>> fitting.LevMarLSQFitter.supported_constraints
['fixed', 'tied', 'bounds']
>>> fitting.SLSQPLSQFitter.supported_constraints
['bounds', 'eqcons', 'ineqcons', 'fixed', 'tied']
```

12.3.4 Defining New Model Classes

This document describes how to add a model to the package or to create a user-defined model. In short, one needs to define all model parameters and write an eval function which evaluates the model. If the model is fittable, a function to compute the derivatives with respect to parameters is required if a linear fitting algorithm is to be used and optional if a non-linear fitter is to be used.

Custom 1-D models

For 1-D models, the `custom_model_1d` decorator is provided to make it very easy to define new models. The following example demonstrates how to set up a model consisting of two Gaussians:

```
import numpy as np
from astropy.modeling.models import custom_model_1d
from astropy.modeling.fitting import LevMarLSQFitter

# Define model
@custom_model_1d
def sum_of_gaussians(x, amplitude1=1., mean1=-1., sigma1=1.,
                    amplitude2=1., mean2=1., sigma2=1.):
    return (amplitude1 * np.exp(-0.5 * ((x - mean1) / sigma1)**2) +
            amplitude2 * np.exp(-0.5 * ((x - mean2) / sigma2)**2))

# Generate fake data
np.random.seed(0)
x = np.linspace(-5., 5., 200)
m_ref = sum_of_gaussians(amplitude1=2., mean1=-0.5, sigma1=0.4,
                        amplitude2=0.5, mean2=2., sigma2=1.0)
y = m_ref(x) + np.random.normal(0., 0.1, x.shape)

# Fit model to data
m_init = sum_of_gaussians()
fit = LevMarLSQFitter()
m = fit(m_init, x, y)

# Plot the data and the best fit
plt.plot(x, y, 'o', color='k')
plt.plot(x, m(x), color='r', lw=2)
```

Note: Currently this shortcut for model definition only works for 1-D models, but it is being expanded to support 2 or greater dimension models.

A step by step definition of a 1-D Gaussian model

The example described in [Custom 1-D models](#) can be used for most 1-D cases, but the following section described how to construct model classes in general. The details are explained below with a 1-D Gaussian model as an example.

There are two base classes for models. If the model is fittable, it should inherit from `FittableModel`; if not it should subclass `Model`.

If the model takes parameters they should be specified as class attributes in the model's class definition using the `Parameter` descriptor. All arguments to the `Parameter` constructor are optional, and may include a default value for that parameter, a text description of the parameter (useful for `help` and documentation generation), as well default constraints and custom getters/setters for the parameter value.

If the first argument `name` is specified it must be identical to the class attribute being assigned that `Parameter`. As such, `Parameters` take their name from this attribute by default. In other words, `amplitude = Parameter('amplitude')` is equivalent to `amplitude = Parameter()`. This differs from Astropy v0.3.x, where it was necessary to provide the name twice.

```
from astropy.modeling import FittableModel, Parameter, format_input
```

```
class Gaussian1DModel(FittableModel):
    amplitude = Parameter()
    mean = Parameter()
    stddev = Parameter()
```

At a minimum, the `__init__` method takes all parameters and a few keyword arguments such as values for constraints:

```
def __init__(self, amplitude, mean, stddev, **kwargs):
    # Note that this __init__ does nothing different from the base class's
    # __init__. The main point of defining it is so that the function
    # signature is more informative.
    super(Gaussian1DModel, self).__init__(
        amplitude=amplitude, mean=mean, stddev=stddev, **kwargs)
```

Note: If a parameter is defined with a default value you may make the argument for that parameter in the `__init__` optional. Otherwise it is recommended to make it a required argument. In the above example none of the parameters have default values.

Fittable models can be linear or nonlinear in a regression sense. The default value of the `linear` attribute is `False`. Linear models should define the `linear` class attribute as `True`. The `n_inputs` attribute stores the number of input variables the model expects. The `n_outputs` attribute stores the number of output variables returned after evaluating the model. These two attributes are used with composite models.

Next, provide methods called `eval` to evaluate the model and `fit_deriv`, to compute its derivatives with respect to parameters. These may be normal methods, `classmethod`, or `staticmethod`, though the convention is to use `staticmethod` when the function does not depend on any of the object's other attributes (i.e., it does not reference `self`). The evaluation method takes all input coordinates as separate arguments and all of the model's parameters in the same order they would be listed by `param_names`.

For this example:

```
@staticmethod
def eval(x, amplitude, mean, stddev):
    return amplitude * np.exp((-1 / (2. * stddev**2)) * (x - mean)**2)
```

The `fit_deriv` method takes as input all coordinates as separate arguments. There is an option to compute numerical derivatives for nonlinear models in which case the `fit_deriv` method should be `None`:

```
@staticmethod
def fit_deriv(x, amplitude, mean, stddev):
    d_amplitude = np.exp((-1 / (stddev**2)) * (x - mean)**2)
    d_mean = (2 * amplitude *
              np.exp((-1 / (stddev**2)) * (x - mean)**2)) *
```

```
        (x - mean) / (stddev**2))
d_stddev = (2 * amplitude *
            np.exp((-1 / (stddev**2)) * (x - mean)**2)) *
            ((x - mean)**2) / (stddev**3))
return [d_amplitude, d_mean, d_stddev]
```

Finally, the `__call__` method takes input coordinates as separate arguments. It reformats them (if necessary) using the `format_input` wrapper/decorator and calls the `eval` method to perform the model evaluation using the input variables and a special property called `param_sets` which returns a list of all the parameter values over all models in the set.

The reason there is a separate `eval` method is to allow fitters to call the `eval` method with different parameters which is necessary for updating the approximation while fitting, and for fitting with constraints.:

```
@format_input
def __call__(self, x):
    return self.eval(x, *self.param_sets)
```

Full example

```
from astropy.modeling import FittableModel, Parameter, format_input

class Gaussian1DModel(FittableModel):
    amplitude = Parameter()
    mean = Parameter()
    stddev = Parameter()

    def __init__(self, amplitude, mean, stddev, **kwargs):
        # Note that this __init__ does nothing different from the base class's
        # __init__. The main point of defining it is so that the function
        # signature is more informative.
        super(Gaussian1DModel, self).__init__(
            amplitude=amplitude, mean=mean, stddev=stddev, **kwargs)

    @staticmethod
    def eval(x, amplitude, mean, stddev):
        return amplitude * np.exp((-1 / (2. * stddev**2)) * (x - mean)**2))

    @staticmethod
    def fit_deriv(x, amplitude, mean, stddev):
        d_amplitude = np.exp((-1 / (stddev**2)) * (x - mean)**2)
        d_mean = (2 * amplitude *
                  np.exp((-1 / (stddev**2)) * (x - mean)**2)) *
                  (x - mean) / (stddev**2)
        d_stddev = (2 * amplitude *
                    np.exp((-1 / (stddev**2)) * (x - mean)**2)) *
                    ((x - mean)**2) / (stddev**3)
        return [d_amplitude, d_mean, d_stddev]

    @format_input
    def __call__(self, x):
        return self.eval(x, *self.param_sets)
```

A full example of a LineModel

```

from astropy.modeling import models, Parameter, format_input
import numpy as np

class LineModel(models.PolynomialModel):
    slope = Parameter()
    intercept = Parameter()
    linear = True

def __init__(self, slope, intercept, **kwargs):
    super(LineModel, self).__init__(slope=slope, intercept=intercept,
                                     **kwargs)

@staticmethod
def eval(x, slope, intercept):
    return slope * x + intercept

@staticmethod
def fit_deriv(x, slope, intercept):
    d_slope = x
    d_intercept = np.ones_like(x)
    return [d_slope, d_intercept]

@format_input
def __call__(self, x):
    return self.eval(x, *self.param_sets)

```

12.3.5 Defining New Fitter Classes

This section describes how to add a new nonlinear fitting algorithm to this package or write a user-defined fitter. In short, one needs to define an error function and a `__call__` method and define the types of constraints which work with this fitter (if any).

The details are described below using `scipy`'s SLSQP algorithm as an example. The base class for all fitters is `Fitter`:

```

class SLSQPFitter(Fitter):
    supported_constraints = ['bounds', 'eqcons', 'ineqcons', 'fixed',
                           'tied']

    def __init__(self):
        # Most currently defined fitters take no arguments in their
        # __init__, but the option certainly exists for custom fitters
        super(SLSQPFitter, self).__init__()

```

All fitters take a model (their `__call__` method modifies the model's parameters) as their first argument.

Next, the error function takes a list of parameters returned by an iteration of the fitting algorithm and input coordinates, evaluates the model with them and returns some type of a measure for the fit. In the example the sum of the squared residuals is used as a measure of fitting.:

```

def objective_function(self, fps, *args):
    model = args[0]
    meas = args[-1]
    model.fitparams(fps)
    res = self.model(*args[1:-1]) - meas
    return np.sum(res**2)

```

The `__call__` method performs the fitting. As a minimum it takes all coordinates as separate arguments. Additional arguments are passed as necessary.:

```
def __call__(self, model, x, y, maxiter=MAXITER, epsilon=EPS):
    if model.linear:
        raise ModelLinearityException(
            'Model is linear in parameters; '
            'non-linear fitting methods should not be used.')
    model_copy = model.copy()
    init_values, _ = _model_to_fit_params(model_copy)
    self.fitparams = optimize.fmin_slsqp(self.errorfunc, p0=init_values,
                                       args=(y, x),
                                       bounds=self.bounds,
                                       eqcons=self.eqcons,
                                       ineqcons=self.ineqcons)

    return model_copy
```

12.3.6 Using a Custom Statistic Function

This section describes how to write a new fitter with a user-defined statistic function. The example below shows a specialized class which fits a straight line with uncertainties in both variables.

The following import statements are needed.:

```
import numpy as np
from astropy.modeling.fitting import (_validate_model,
                                     _fitter_to_model_params,
                                     _model_to_fit_params, Fitter,
                                     _convert_input)
from astropy.modeling.optimizers import Simplex
```

First one needs to define a statistic. This can be a function or a callable class.:

```
def chi_line(measured_vals, updated_model, x_sigma, y_sigma, x):
    """
    Chi2 statistic for fitting a straight line with uncertainties in x and
    y.

    Parameters
    -----
    measured_vals : array
    updated_model : `~astropy.modeling.ParametricModel`
        model with parameters set by the current iteration of the optimizer
    x_sigma : array
        uncertainties in x
    y_sigma : array
        uncertainties in y

    """
    model_vals = updated_model(x)
    if x_sigma is None and y_sigma is None:
        return np.sum((model_vals - measured_vals) ** 2)
    elif x_sigma is not None and y_sigma is not None:
        weights = 1 / (y_sigma ** 2 + updated_model.parameters[1] ** 2 *
                      x_sigma ** 2)
        return np.sum((weights * (model_vals - measured_vals)) ** 2)
    else:
        if x_sigma is not None:
```

```

        weights = 1 / x_sigma ** 2
    else:
        weights = 1 / y_sigma ** 2
    return np.sum((weights * (model_vals - measured_vals)) ** 2)

```

In general, to define a new fitter, all one needs to do is provide a statistic function and an optimizer. In this example we will let the optimizer be an optional argument to the fitter and will set the statistic to `chi_line` above.:

```

class LineFitter(Fitter):
    """
    Fit a straight line with uncertainties in both variables

    Parameters
    -----
    optimizer : class or callable
        one of the classes in optimizers.py (default: Simplex)
    """

    def __init__(self, optimizer=Simplex):
        self.statistic = chi_line
        super(LineFitter, self).__init__(optimizer,
                                         statistic=self.statistic)

```

The last thing to define is the `__call__` method.:

```

def __call__(self, model, x, y, x_sigma=None, y_sigma=None, **kwargs):
    """
    Fit data to this model.

    Parameters
    -----
    model : `~astropy.modeling.core.ParametricModel`
        model to fit to x, y
    x : array
        input coordinates
    y : array
        input coordinates
    x_sigma : array
        uncertainties in x
    y_sigma : array
        uncertainties in y
    kwargs : dict
        optional keyword arguments to be passed to the optimizer

    Returns
    -----
    model_copy : `~astropy.modeling.core.ParametricModel`
        a copy of the input model with parameters set by the fitter
    """
    model_copy = _validate_model(model,
                                  self._opt_method.supported_constraints)

    farg = _convert_input(x, y)
    farg = (model_copy, x_sigma, y_sigma) + farg
    p0, _ = _model_to_fit_params(model_copy)

    fitparams, self.fit_info = self._opt_method(
        self.objective_function, p0, farg, **kwargs)

```

```
_fitter_to_model_params(model_copy, fitparams)

return model_copy
```

12.3.7 Algorithms

Univariate polynomial evaluation

- The evaluation of 1-D polynomials uses Horner’s algorithm.
- The evaluation of 1-D Chebyshev and Legendre polynomials uses Clenshaw’s algorithm.

Multivariate polynomial evaluation

- Multivariate Polynomials are evaluated following the algorithm in ¹. The algorithm uses the following notation:
 - **multiindex** is a tuple of non-negative integers for which the length is defined in the following way:

$$\alpha = (\alpha_1, \alpha_2, \alpha_3), |\alpha| = \alpha_1 + \alpha_2 + \alpha_3$$

- **inverse lexical order** is the ordering of monomials in such a way that $x^a < x^b$ if and only if there exists $1 \leq i \leq n$ such that $a_n = b_n, \dots, a_{i+1} = b_{i+1}, a_i < b_i$.

In this ordering $y^2 > x^2 * y$ and $x * y > y$

- **Multivariate Horner scheme** uses $d+1$ variables r_0, \dots, r_d to store intermediate results, where d denotes the number of variables.

Algorithm:

1. Set d_i to the max number of variables (2 for a 2-D polynomials).
 2. Set r_0 to $c_{\alpha(0)}$, where c is a list of coefficients for each multiindex in inverse lexical order.
 3. For each monomial, n , in the polynomial:
 - * determine $k = \max\{1 \leq j \leq d_i : \alpha(n)_j \neq \alpha(n-1)_j\}$
 - * Set $r_k := l_k(x) * (r_0 + r_1 + \dots + r_k)$
 - * Set $r_0 = c_{\alpha(n)}, r_1 = \dots r_{k-1} = 0$.
 4. return $r_0 + \dots + r_{d_i}$
- The evaluation of multivariate Chebyshev and Legendre polynomials uses a variation of the above Horner’s scheme, in which every Legendre or Chebyshev function is considered a separate variable. In this case the length of the α indices tuple is equal to the number of functions in x plus the number of functions in y . In addition the Chebyshev and Legendre functions are cached for efficiency.

12.3.8 Models Design Goals

The `astropy.modeling` and `astropy.modeling.fitting` modules described here are designed to work as peers. The goal is to be able to add models without explicit reference to fitting algorithms and likewise, add different fitting algorithms without changing the existing models.

¹

10. (a) Pena, Thomas Sauer, “On the Multivariate Horner Scheme”, SIAM Journal on Numerical Analysis, Vol 37, No. 4

Furthermore, the models are designed to be combined in many ways. It is possible, for example, to combine models `serially` so that the output values of one model are used as input values to another. It is also possible to form a new model by combining models in `parallel` (each model is evaluated separately with the original input and the deltas are summed). Since models may have multiple input values, machinery is provided that allows assigning outputs from one model into the appropriate input of another in a flexible way, `LabeledInput`. Finally, it is permitted to combine any number of models using all of these mechanisms simultaneously. A composite model can be used to make further composite models.

In the future this will support a model language which will allow using models in algebraic operations like

$$model = (model_1 + model_2) * model_3$$

12.4 Reference/API

12.4.1 astropy.modeling Module

This subpackage provides a framework for representing models and performing model evaluation and fitting. It supports 1D and 2D models and fitting with parameter constraints. It has some predefined models and fitting routines.

Functions

`format_input(func)` Wraps a model's `__call__` method so that the input arrays are converted into the appropriate shape given

`format_input`

`astropy.modeling.format_input(func)`

Wraps a model's `__call__` method so that the input arrays are converted into the appropriate shape given the model's parameter dimensions.

Wraps the result to match the shape of the last input array.

Classes

<code>Fittable1DModel(*args, **kwargs)</code>	Base class for one-dimensional fittable models.
<code>Fittable2DModel(*args, **kwargs)</code>	Base class for one-dimensional fittable models.
<code>FittableModel(*args, **kwargs)</code>	Base class for models that can be fitted using the built-in fitting algorithm.
<code>InputParameterError</code>	Used for incorrect input parameter values and definitions.
<code>LabeledInput(data, labels)</code>	Used by <code>SerialCompositeModel</code> and <code>SummedCompositeModel</code> .
<code>Model(*args, **kwargs)</code>	Base class for all models.
<code>ModelDefinitionError</code>	Used for incorrect models definitions.
<code>Parameter([name, description, default, ...])</code>	Wraps individual parameters.
<code>SerialCompositeModel(transforms[, inmap, ...])</code>	Composite model that evaluates models in series.
<code>SummedCompositeModel(transforms[, inmap, outmap])</code>	Composite model that evaluates models in parallel.

`Fittable1DModel`

`class astropy.modeling.Fittable1DModel(*args, **kwargs)`

Bases: `astropy.modeling.FittableModel`

Base class for one-dimensional fittable models.

This class provides an easier interface to defining new models. Examples can be found in `astropy.modeling.functional_models`.

Methods Summary

<code>__call__(*inputs, **kwargs)</code>	Transforms data using this model.
<code>eval()</code>	A method, <code>classmethod</code> , or <code>staticmethod</code> that implements evaluation of the function repres

Methods Documentation

`__call__(*inputs, **kwargs)`
Transforms data using this model.

Parameters

x : array-like or numeric value

Input coordinate values.

model_set_axis : `int` or `False`, optional

For `Model` instances representing a multiple-model set, this picks out which axis of the input array is used to map inputs to specific models in the set. If `False`, this indicates that the input array has no such axis, and instead the same input should be broadcast to all models in the set.

`eval()`
A method, `classmethod`, or `staticmethod` that implements evaluation of the function represented by this model.

It must take arguments of the function's independent variables, followed by the function's parameters given in the same order they are listed by `Model.param_names`.

Fittable2DModel

`class astropy.modeling.Fittable2DModel(*args, **kwargs)`

Bases: `astropy.modeling.FittableModel`

Base class for one-dimensional fittable models.

This class provides an easier interface to defining new models. Examples can be found in `astropy.modeling.functional_models`.

Attributes Summary

<code>n_inputs</code>	<code>int(x[, base]) -> integer</code>
<code>n_outputs</code>	<code>int(x[, base]) -> integer</code>

Methods Summary

Conti

Table 12.5 – continued from previous page

<code>__call__(*inputs, **kwargs)</code>	Transforms data using this model.
<code>eval()</code>	A method, <code>classmethod</code> , or <code>staticmethod</code> that implements evaluation of the function repres

Attributes Documentation**n_inputs = 2****n_outputs = 1****Methods Documentation**

`__call__(*inputs, **kwargs)`
Transforms data using this model.

Parameters**x** : array-like or numeric value

First input coordinate values.

y : array-like or numeric value

Second input coordinate values.

model_set_axis : `int` or `False`, optional

For `Model` instances representing a multiple-model set, this picks out which axis of the input array is used to map inputs to specific models in the set. If `False`, this indicates that the input array has no such axis, and instead the same input should be broadcast to all models in the set.

eval()

A method, `classmethod`, or `staticmethod` that implements evaluation of the function represented by this model.

It must take arguments of the function's independent variables, followed by the function's parameters given in the same order they are listed by `Model.param_names`.

FittableModel

class `astropy.modeling.FittableModel(*args, **kwargs)`

Bases: `astropy.modeling.Model`

Base class for models that can be fitted using the built-in fitting algorithms.

Attributes Summary

<code>col_fit_deriv</code>	<code>bool(x) -> bool</code>
<code>fit_deriv</code>	
<code>fittable</code>	<code>bool(x) -> bool</code>
<code>linear</code>	<code>bool(x) -> bool</code>

Attributes Documentation

col_fit_deriv = True

fit_deriv = None

Function (similar to the model's `eval`) to compute the derivatives of the model with respect to its parameters, for use by fitting algorithms.

fittable = True

linear = False

InputParameterError

exception `astropy.modeling.InputParameterError`

Used for incorrect input parameter values and definitions.

LabeledInput

class `astropy.modeling.LabeledInput` (*data, labels*)

Bases: `dict`

Used by `SerialCompositeModel` and `SummedCompositeModel` to choose input data using labels.

This is a container assigning labels (names) to all input data arrays to a composite model.

Parameters

data : list

List of all input data

labels : list of strings

names matching each coordinate in data

Examples

```
>>> y, x = np.mgrid[:5, :5]
>>> l = np.arange(10)
>>> labeled_input = LabeledInput([x, y, l], ['x', 'y', 'pixel'])
>>> labeled_input.x
array([[0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4]])
>>> labeled_input['x']
array([[0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4]])
```

Methods Summary

<code>add([label, value])</code>	Add input data to a LabeledInput object
<code>copy()</code>	

Methods Documentation

add (*label=None, value=None, **kw*)

Add input data to a LabeledInput object

Parameters

label : str

coordinate label

value : numerical type

coordinate value

kw : dictionary

if given this is a dictionary of {label: value} pairs

copy ()

Model

class `astropy.modeling.Model` (**args, **kwargs*)

Bases: `object`

Base class for all models.

This is an abstract class and should not be instantiated directly.

This class sets the constraints and other properties for all individual parameters and performs parameter validation.

Parameters

param_dim : int

Number of parameter sets

fixed : dict

Dictionary {parameter_name: bool} setting the fixed constraint for one or more parameters. `True` means the parameter is held fixed during fitting and is prevented from updates once an instance of the model has been created.

Alternatively the `fixed` property of a parameter may be used to lock or unlock individual parameters.

tied : dict

Dictionary {parameter_name: callable} of parameters which are linked to some other parameter. The dictionary values are callables providing the linking relationship.

Alternatively the `tied` property of a parameter may be used to set the `tied` constraint on individual parameters.

bounds : dict

Dictionary {parameter_name: value} of lower and upper bounds of parameters. Keys are parameter names. Values are a list of length 2 giving the desired range for the parameter.

Alternatively the `min` and `max` or `~astropy.modeling.Parameter.bounds` properties of a parameter may be used to set bounds on individual parameters.

eqcons : list

List of functions of length n such that `eqcons[j](x0, *args) == 0.0` in a successfully optimized problem.

ineqcons : list

List of functions of length n such that `ineqcons[j](x0, *args) >= 0.0` is a successfully optimized problem.

Examples

```
>>> from astropy.modeling import models
>>> def tie_center(model):
...     mean = 50 * model.stddev
...     return mean
>>> tied_parameters = {'mean': tie_center}
```

Specify that 'mean' is a tied parameter in one of two ways:

```
>>> g1 = models.Gaussian1D(amplitude=10, mean=5, stddev=.3,
...                         tied=tied_parameters)
```

or

```
>>> g1 = models.Gaussian1D(amplitude=10, mean=5, stddev=.3)
>>> g1.mean.tied
False
>>> g1.mean.tied = tie_center
>>> g1.mean.tied
<function tie_center at 0x...>
```

Fixed parameters:

```
>>> g1 = models.Gaussian1D(amplitude=10, mean=5, stddev=.3,
...                         fixed={'stddev': True})
>>> g1.stddev.fixed
True
```

or

```
>>> g1 = models.Gaussian1D(amplitude=10, mean=5, stddev=.3)
>>> g1.stddev.fixed
False
>>> g1.stddev.fixed = True
>>> g1.stddev.fixed
True
```

Attributes Summary

<code>bounds</code>	A <code>dict</code> mapping parameter names to their upper and lower bounds as <code>(min, max)</code> tuples.
<code>eqcons</code>	List of parameter equality constraints.
<code>fittable</code>	<code>bool(x) -> bool</code>
<code>fixed</code>	A <code>dict</code> mapping parameter names to their fixed constraint.
<code>ineqcons</code>	List of parameter inequality constraints.
<code>linear</code>	<code>bool(x) -> bool</code>
<code>model_constraints</code>	<code>list() -> new empty list</code>
<code>model_set_axis</code>	
<code>n_inputs</code>	<code>int(x[, base]) -> integer</code>
<code>n_outputs</code>	<code>int(x[, base]) -> integer</code>
<code>param_dim</code>	Deprecated since version 0.4.
<code>param_names</code>	<code>list() -> new empty list</code>
<code>param_sets</code>	Return parameters as a pset.
<code>parameter_constraints</code>	<code>list() -> new empty list</code>
<code>parameters</code>	A flattened array of all parameter values in all parameter sets.
<code>standard_broadcasting</code>	<code>bool(x) -> bool</code>
<code>tied</code>	A <code>dict</code> mapping parameter names to their tied constraint.

Methods Summary

<code>__call__(*args, **kwargs)</code>	Evaluate the model on some input variables.
<code>add_model(model, mode)</code>	Create a CompositeModel by chaining the current model with the new one using the specified mode.
<code>copy()</code>	Return a copy of this model.
<code>inverse()</code>	Returns a callable object which performs the inverse transform.
<code>invert()</code>	Invert coordinates iteratively if possible.

Attributes Documentation

bounds

A `dict` mapping parameter names to their upper and lower bounds as `(min, max)` tuples.

eqcons

List of parameter equality constraints.

fittable = False

fixed

A `dict` mapping parameter names to their fixed constraint.

ineqcons

List of parameter inequality constraints.

linear = True

`model_constraints = [u'eqcons', u'ineqcons']`

`model_set_axis`

`n_inputs = 1`

n_outputs = 1

param_dim

Deprecated since version 0.4: The param_dim function is deprecated and may be removed in a future version. Use len(model) instead.

param_names = []

List of names of the parameters that describe models of this type.

The parameters in this list are in the same order they should be passed in when initializing a model of a specific type. Some types of models, such as polynomial models, have a different number of parameters depending on some other property of the model, such as the degree.

param_sets

Return parameters as a pset.

This is an array where each column represents one parameter set.

parameter_constraints = [u'fixed', u'tied', u'bounds']

parameters

A flattened array of all parameter values in all parameter sets.

Fittable parameters maintain this list and fitters modify it.

standard_broadcasting = True

tied

A dict mapping parameter names to their tied constraint.

Methods Documentation

__call__ (*args, **kwargs)

Evaluate the model on some input variables.

add_model (model, mode)

Create a CompositeModel by chaining the current model with the new one using the specified mode.

Parameters

model : an instance of a subclass of Model

mode : string

'parallel', 'serial', 'p' or 's' a flag indicating whether to combine the models in series or in parallel

Returns

model : CompositeModel

an instance of CompositeModel

copy ()

Return a copy of this model.

Uses a deep copy so that all model attributes, including parameter values, are copied as well.

inverse ()

Returns a callable object which performs the inverse transform.

invert ()

Invert coordinates iteratively if possible.

ModelDefinitionError

exception `astropy.modeling.ModelDefinitionError`

Used for incorrect models definitions

Parameter

class `astropy.modeling.Parameter` (*name='u', description='u', default=None, getter=None, setter=None, fixed=False, tied=False, min=None, max=None, model=None*)

Bases: `object`

Wraps individual parameters.

This class represents a model’s parameter (in a somewhat broad sense). It acts as both a descriptor that can be assigned to a class attribute to describe the parameters accepted by an individual model (this is called an “unbound parameter”), or it can act as a proxy for the parameter values on an individual model instance (called a “bound parameter”).

Parameter instances never store the actual value of the parameter directly. Rather, each instance of a model stores its own parameters as either hidden attributes or (in the case of `FittableModel`) in an array. A *bound* Parameter simply wraps the value in a Parameter proxy which provides some additional information about the parameter such as its constraints.

Unbound Parameters are not associated with any specific model instance, and are merely used by model classes to determine the names of their parameters and other information about each parameter such as their default values and default constraints.

Parameters

name : str

parameter name

description : str

parameter description

default : float or array

default value to use for this parameter

getter : callable

a function that wraps the raw (internal) value of the parameter when returning the value through the parameter proxy (eg. a parameter may be stored internally as radians but returned to the user as degrees)

setter : callable

a function that wraps any values assigned to this parameter; should be the inverse of getter

fixed : bool

if True the parameter is not varied during fitting

tied : callable or False

if callable is supplied it provides a way to link the value of this parameter to another parameter (or some other arbitrary function)

min : float

the lower bound of a parameter

max : float

the upper bound of a parameter

model : object

an instance of a Model class; this should only be used internally for creating bound Parameters

Attributes Summary

<code>bounds</code>	The minimum and maximum values of a parameter as a tuple
<code>default</code>	Parameter default value
<code>fixed</code>	Boolean indicating if the parameter is kept fixed during fitting.
<code>max</code>	A value used as an upper bound when fitting a parameter
<code>min</code>	A value used as a lower bound when fitting a parameter
<code>name</code>	Parameter name
<code>shape</code>	The shape of this parameter's value array.
<code>size</code>	The size of this parameter's value array.
<code>tied</code>	Indicates that this parameter is linked to another one.
<code>value</code>	The unadorned value proxied by this parameter

Attributes Documentation

bounds

The minimum and maximum values of a parameter as a tuple

default

Parameter default value

fixed

Boolean indicating if the parameter is kept fixed during fitting.

max

A value used as an upper bound when fitting a parameter

min

A value used as a lower bound when fitting a parameter

name

Parameter name

shape

The shape of this parameter's value array.

size

The size of this parameter's value array.

tied

Indicates that this parameter is linked to another one.

A callable which provides the relationship of the two parameters.

value

The unadorned value proxied by this parameter

SerialCompositeModel

```
class astropy.modeling.SerialCompositeModel (transforms, inmap=None, outmap=None,
                                             n_inputs=None, n_outputs=None)
```

Bases: `astropy.modeling.core._CompositeModel`

Composite model that evaluates models in series.

Parameters

transforms : list

a list of transforms in the order to be executed

inmap : list of lists or None

labels in an input instance of `LabeledInput` if None, the number of input coordinates is exactly what the transforms expect

outmap : list or None

labels in an input instance of `LabeledInput` if None, the number of output coordinates is exactly what the transforms expect

n_inputs : int

dimension of input space (e.g. 2 for a spatial model)

n_outputs : int

dimension of output

Notes

Output values of one model are used as input values of another. Obviously the order of the models matters.

Examples

Apply a 2D rotation followed by a shift in x and y:

```
>>> import numpy as np
>>> from astropy.modeling import models, LabeledInput, SerialCompositeModel
>>> y, x = np.mgrid[:5, :5]
>>> rotation = models.Rotation2D(angle=23.5)
>>> offset_x = models.Shift(-4.23)
>>> offset_y = models.Shift(2)
>>> labeled_input = LabeledInput([x, y], ["x", "y"])
>>> transform = SerialCompositeModel([rotation, offset_x, offset_y],
...                                 inmap=[['x', 'y'], ['x'], ['y']],
...                                 outmap=[['x', 'y'], ['x'], ['y']])
>>> result = transform(labeled_input)
```

Methods Summary

```
__call__(*data)  Transforms data using this model.  
inverse()
```

Methods Documentation

```
__call__(*data)  
    Transforms data using this model.  
  
inverse()
```

SummedCompositeModel

```
class astropy.modeling.SummedCompositeModel (transforms, inmap=None, outmap=None)  
    Bases: astropy.modeling.core._CompositeModel
```

Composite model that evaluates models in parallel.

Parameters

transforms : list
transforms to be executed in parallel

inmap : list or None
labels in an input instance of LabeledInput if None, the number of input coordinates is exactly what the transforms expect

outmap : list or None

Notes

Evaluate each model separately and add the results to the `input_data`.

Methods Summary

```
__call__(*data)  Transforms data using this model.
```

Methods Documentation

```
__call__(*data)  
    Transforms data using this model.
```

Class Inheritance Diagram

12.4.2 astropy.modeling.fitting Module

This module implements classes (called Fitters) which combine optimization algorithms (typically from `scipy.optimize`) with statistic functions to perform fitting. Fitters are implemented as callable classes. In addition to the data to fit, the `__call__` method takes an instance of `FittableModel` as input, and returns a copy of the model with its parameters determined by the optimizer.

Optimization algorithms, called “optimizers” are implemented in `optimizers` and statistic functions are in `statistic`. The goal is to provide an easy to extend framework and allow users to easily create new fitters by combining statistics with optimizers.

There are two exceptions to the above scheme. `LinearLSQFitter` uses Numpy’s `lstsq` function. `LevMarLSQFitter` uses `leastsq` which combines optimization and statistic in one implementation.

Classes

<code>LinearLSQFitter()</code>	A class performing a linear least square fitting.
<code>LevMarLSQFitter()</code>	Levenberg-Marquardt algorithm and least squares statistic.
<code>SLSQPLSQFitter()</code>	SLSQP optimization algorithm and least squares statistic.
<code>SimplexLSQFitter()</code>	Simplex algorithm and least squares statistic.
<code>JointFitter(models, jointparameters, initvals)</code>	Fit models which share a parameter.
<code>Fitter(optimizer, statistic)</code>	Base class for all fitters.

LinearLSQFitter

class `astropy.modeling.fitting.LinearLSQFitter`

Bases: `object`

A class performing a linear least square fitting.

Uses `numpy.linalg.lstsq` to do the fitting. Given a model and data, fits the model to the data and changes the model’s parameters. Keeps a dictionary of auxiliary fitting information.

Attributes Summary

<code>supported_constraints</code>	<code>list()</code> -> new empty list
------------------------------------	---------------------------------------

Methods Summary

<code>__call__(model, x, y[, z, weights, rcond])</code>	Fit data to this model.
---	-------------------------

Attributes Documentation

`supported_constraints = [u’fixed’]`

Methods Documentation

`__call__` (*model*, *x*, *y*, *z=None*, *weights=None*, *rcond=None*)
Fit data to this model.

Parameters

`model` : `FittableModel`

model to fit to *x*, *y*, *z*

`x` : array

input coordinates
y : array
input coordinates
z : array (optional)
input coordinates
weights : array (optional)
weights
rcond : float, optional
Cut-off ratio for small singular values of a . Singular values are set to zero if they are smaller than `rcond` times the largest singular value of a .

Returns

model_copy : `FittableModel`
a copy of the input model with parameters set by the fitter

LevMarLSQFitter

class `astropy.modeling.fitting.LevMarLSQFitter`

Bases: `object`

Levenberg-Marquardt algorithm and least squares statistic.

Notes

The `fit_info` dictionary contains the values returned by `scipy.optimize.leastsq` for the most recent fit, including the values from the `infodict` dictionary it returns. See the `scipy.optimize.leastsq` documentation for details on the meaning of these values. Note that the `x` return value is *not* included (as it is instead the parameter values of the returned model).

Additionally, one additional element of `fit_info` is computed whenever a model is fit, with the key 'param_cov'. The corresponding value is the covariance matrix of the parameters as a 2D numpy array. The order of the matrix elements matches the order of the parameters in the fitted model (i.e., the same order as `model.param_names`).

Attributes

<code>fit_info</code>	(dict) The <code>scipy.optimize.leastsq</code> result for the most recent fit (see notes).
-----------------------	--

Attributes Summary

<code>supported_constraints</code>	<code>list()</code> -> new empty list
------------------------------------	---------------------------------------

Methods Summary

Continued on next page

Table 12.17 – continued from previous page

<code>__call__(model, x, y[, z, weights, maxiter, ...])</code>	Fit data to this model.
<code>objective_function(fps, *args)</code>	Function to minimize.

Attributes Documentation**supported_constraints = [u'fixed', u'tied', u'bounds']**

The constraint types supported by this fitter type.

Methods Documentation

`__call__(model, x, y, z=None, weights=None, maxiter=100, acc=9.999999999999995e-08, epsilon=1.4901161193847656e-08, estimate_jacobian=False)`

Fit data to this model.

Parameters**model** : `FittableModel`

model to fit to x, y, z

x : array

input coordinates

y : array

input coordinates

z : array (optional)

input coordinates

weights : array (optional)

weights

maxiter : int

maximum number of iterations

acc : float

Relative error desired in the approximate solution

epsilon : floatA suitable step length for the forward-difference approximation of the Jacobian (if `model.fjac=None`). If `epsfcn` is less than the machine precision, it is assumed that the relative errors in the functions are of the order of the machine precision.**estimate_jacobian** : boolIf False (default) and if the model has a `fit_deriv` method, it will be used. Otherwise the Jacobian will be estimated. If True, the Jacobian will be estimated in any case.**Returns****model_copy** : `FittableModel`

a copy of the input model with parameters set by the fitter

objective_function (`fps, *args`)

Function to minimize.

Parameters

fps : list

parameters returned by the fitter

args : list

[model, [weights], [input coordinates]]

SLSQPLSQFitter

class `astropy.modeling.fitting.SLSQPLSQFitter`

Bases: `astropy.modeling.fitting.Fitter`

SLSQP optimization algorithm and least squares statistic.

Raises

ModelLinearityError

A linear model is passed to a nonlinear fitter

Attributes Summary

`supported_constraints` list() -> new empty list

Methods Summary

`__call__(model, x, y[, z, weights])` Fit data to this model.

Attributes Documentation

`supported_constraints = [u'bounds', u'eqcons', u'ineqcons', u'fixed', u'tied']`

Methods Documentation

`__call__(model, x, y, z=None, weights=None, **kwargs)`

Fit data to this model.

Parameters

model : `FittableModel`

model to fit to x, y, z

x : array

input coordinates

y : array

input coordinates

z : array (optional)

input coordinates

weights : array (optional)

weights

kwargs : dict

optional keyword arguments to be passed to the optimizer or the statistic

verblevel : int

0-silent 1-print summary upon completion, 2-print summary after each iteration

maxiter : int

maximum number of iterations

epsilon : float

the step size for finite-difference derivative estimates

acc : float

Requested accuracy

Returns

model_copy : `FittableModel`

a copy of the input model with parameters set by the fitter

SimplexLSQFitter

class `astropy.modeling.fitting.SimplexLSQFitter`

Bases: `astropy.modeling.fitting.Fitter`

Simplex algorithm and least squares statistic.

Raises

ModelLinearityError

A linear model is passed to a nonlinear fitter

Attributes Summary

`supported_constraints` list() -> new empty list

Methods Summary

`__call__(model, x, y[, z, weights])` Fit data to this model.

Attributes Documentation

supported_constraints = ['u'bounds', 'u'fixed', 'u'tied']

Methods Documentation

`__call__` (*model*, *x*, *y*, *z=None*, *weights=None*, ***kwargs*)

Fit data to this model.

Parameters

model : `FittableModel`

model to fit to *x*, *y*, *z*

x : array

input coordinates

y : array

input coordinates

z : array (optional)

input coordinates

weights : array (optional)

weights

kwargs : dict

optional keyword arguments to be passed to the optimizer or the statistic

maxiter : int

maximum number of iterations

epsilon : float

the step size for finite-difference derivative estimates

acc : float

Relative error in approximate solution

Returns

model_copy : `FittableModel`

a copy of the input model with parameters set by the fitter

JointFitter

class `astropy.modeling.fitting.JointFitter` (*models*, *jointparameters*, *initvals*)

Bases: `object`

Fit models which share a parameter.

For example, fit two gaussians to two data sets but keep the FWHM the same.

Parameters

models : list

a list of model instances

jointparameters : list

a list of joint parameters

initvals : list

a list of initial values

Methods Summary

<code>__call__(*args)</code>	Fit data to these models keeping some of the parameters common to the two models.
<code>objective_function(fps, *args)</code>	Function to minimize.

Methods Documentation

`__call__(*args)`

Fit data to these models keeping some of the parameters common to the two models.

`objective_function(fps, *args)`

Function to minimize.

Parameters

fps : list

the fitted parameters - result of an one iteration of the fitting algorithm

args : dict

tuple of measured and input coordinates args is always passed as a tuple from `optimize.leastsq`

Fitter

class `astropy.modeling.fitting.Fitter` (*optimizer, statistic*)

Bases: `object`

Base class for all fitters.

Parameters

optimizer : callable

A callable implementing an optimization algorithm

statistic : callable

Statistic function

Methods Summary

<code>__call__()</code>	This method performs the actual fitting and modifies the parameter list of a model.
<code>objective_function(fps, *args)</code>	Function to minimize.

Methods Documentation

`__call__()`

This method performs the actual fitting and modifies the parameter list of a model.

Fitter subclasses should implement this method.

`objective_function(fps, *args)`

Function to minimize.

Parameters

fps : list

parameters returned by the fitter

args : list

[model, [other_args], [input coordinates]] other_args may include weights or any other quantities specific for a statistic

Notes

The list of arguments (args) is set in the `__call__` method. Fitters may overwrite this method, e.g. when statistic functions require other arguments.

Class Inheritance Diagram

12.4.3 astropy.modeling.functional_models Module

Mathematical models.

Functions

`custom_model_1d(func[, func_fit_deriv])` Create a one dimensional model from a user defined function.

custom_model_1d

`astropy.modeling.functional_models.custom_model_1d(func, func_fit_deriv=None)`

Create a one dimensional model from a user defined function. The parameters of the model will be inferred from the arguments of the function.

Note: All model parameters have to be defined as keyword arguments with default values in the model function.

If you want to use parameter sets in the model, the parameters should be treated as lists or arrays.

Parameters

func : function

Function which defines the model. It should take one positional argument (the independent variable in the model), and any number of keyword arguments (the parameters). It must return the value of the model (typically as an array, but can also be a scalar for scalar inputs). This corresponds to the `eval` method.

func_fit_deriv : function, optional

Function which defines the Jacobian derivative of the model. I.e., the derivative with respect to the *parameters* of the model. It should have the same argument signature as `func`, but should return a sequence where each element of the sequence is the derivative with respect to the corresponding argument. This corresponds to the `fit_deriv()` method.

Examples

Define a sinusoidal model function as a custom 1D model:

```
>>> from astropy.modeling.models import custom_model_1d
>>> import numpy as np
>>> def sine_model(x, amplitude=1., frequency=1.):
...     return amplitude * np.sin(2 * np.pi * frequency * x)
>>> def sine_deriv(x, amplitude=1., frequency=1.):
...     return 2 * np.pi * amplitude * np.cos(2 * np.pi * frequency * x)
>>> SineModel = custom_model_1d(sine_model, func_fit_deriv=sine_deriv)
```

Create an instance of the custom model and evaluate it:

```
>>> model = SineModel()
>>> model(0.25)
1.0
```

This model instance can now be used like a usual astropy model.

Classes

<code>AiryDisk2D(amplitude, x_0, y_0, radius, **kwargs)</code>	Two dimensional Airy disk model.
<code>Beta1D(amplitude, x_0, gamma, alpha, **kwargs)</code>	One dimensional Beta model.
<code>Beta2D(amplitude, x_0, y_0, gamma, alpha, ...)</code>	Two dimensional Beta model.
<code>Box1D(amplitude, x_0, width, **kwargs)</code>	One dimensional Box model.
<code>Box2D(amplitude, x_0, y_0, x_width, y_width, ...)</code>	Two dimensional Box model.
<code>Const1D(amplitude, **kwargs)</code>	One dimensional Constant model.
<code>Const2D(amplitude, **kwargs)</code>	Two dimensional Constant model.
<code>Disk2D(amplitude, x_0, y_0, R_0, **kwargs)</code>	Two dimensional radial symmetric Disk model.
<code>Gaussian1D(amplitude, mean, stddev, **kwargs)</code>	One dimensional Gaussian model.
<code>Gaussian2D(amplitude, x_mean, y_mean[, ...])</code>	Two dimensional Gaussian model.
<code>GaussianAbsorption1D(amplitude, mean, ...)</code>	One dimensional Gaussian absorption line model.
<code>Linear1D(slope, intercept, **kwargs)</code>	One dimensional Line model.
<code>Lorentz1D(amplitude, x_0, fwhm, **kwargs)</code>	One dimensional Lorentzian model.
<code>MexicanHat1D(amplitude, x_0, sigma, **kwargs)</code>	One dimensional Mexican Hat model.
<code>MexicanHat2D(amplitude, x_0, y_0, sigma, ...)</code>	Two dimensional symmetric Mexican Hat model.
<code>Redshift(z, **kwargs)</code>	One dimensional redshift model.
<code>Ring2D(amplitude, x_0, y_0, r_in[, width, r_out])</code>	Two dimensional radial symmetric Ring model.
<code>Scale(factors, **kwargs)</code>	Multiply a model by a factor.
<code>Shift(offsets, **kwargs)</code>	Shift a coordinate.
<code>Sine1D(amplitude, frequency, **kwargs)</code>	One dimensional Sine model.
<code>Trapezoid1D(amplitude, x_0, width, slope, ...)</code>	One dimensional Trapezoid model.
<code>TrapezoidDisk2D(amplitude, x_0, y_0, R_0, ...)</code>	Two dimensional circular Trapezoid model.

AiryDisk2D

```
class astropy.modeling.functional_models.AiryDisk2D(amplitude, x_0, y_0, radius,
                                                    **kwargs)
```

Bases: `astropy.modeling.Fittable2DModel`

Two dimensional Airy disk model.

Parameters

amplitude : float

Amplitude of the Airy function.

x_0 : float

x position of the maximum of the Airy function.

y_0 : float

y position of the maximum of the Airy function.

radius : float

The radius of the Airy disk (radius of the first zero).

Other Parameters

fixed : a dict

A dictionary `{parameter_name: boolean}` of parameters to not be varied during fitting. True means the parameter is held fixed. Alternatively the `fixed` property of a parameter may be used.

tied : dict

A dictionary `{parameter_name: callable}` of parameters which are linked to some other parameter. The dictionary values are callables providing the linking relationship. Alternatively the `tied` property of a parameter may be used.

bounds : dict

A dictionary `{parameter_name: boolean}` of lower and upper bounds of parameters. Keys are parameter names. Values are a list of length 2 giving the desired range for the parameter. Alternatively the `min` and `max` properties of a parameter may be used.

eqcons : list

A list of functions of length `n` such that `eqcons[j](x0, *args) == 0.0` in a successfully optimized problem.

ineqcons : list

A list of functions of length `n` such that `ineqcons[j](x0, *args) >= 0.0` is a successfully optimized problem.

See also:

`Box2D`, `TrapezoidDisk2D`, `Gaussian2D`

Notes

Model formula:

$$f(r) = A \left[\frac{2J_1\left(\frac{\pi r}{R/R_z}\right)}{\frac{\pi r}{R/R_z}} \right]^2$$

Where J_1 is the first order Bessel function of the first kind, r is radial distance from the maximum of the Airy function ($r = \sqrt{(x - x_0)^2 + (y - y_0)^2}$), R is the input `radius` parameter, and $R_z = 1.2196698912665045$.

For an optical system, the radius of the first zero represents the limiting angular resolution and is approximately $1.22 * \lambda / D$, where λ is the wavelength of the light and D is the diameter of the aperture.

See [R5] for more details about the Airy disk.

References

[R5]

Attributes Summary

```

amplitude
param_names  list() -> new empty list
radius
x_0
y_0

```

Methods Summary

```

eval(x, y, amplitude, x_0, y_0, radius)  Two dimensional Airy model function

```

Attributes Documentation

amplitude

param_names = ['amplitude', 'x_0', 'y_0', 'radius']

radius

x_0

y_0

Methods Documentation

classmethod eval (*x*, *y*, *amplitude*, *x_0*, *y_0*, *radius*)

Two dimensional Airy model function

Beta1D

class `astropy.modeling.functional_models.Beta1D` (*amplitude*, *x_0*, *gamma*, *alpha*, ***kwargs*)

Bases: `astropy.modeling.Fittable1DModel`

One dimensional Beta model.

Parameters

amplitude : float

Amplitude of the model.

x_0 : float

x position of the maximum of the Beta model.

gamma : float

Core width of the Beta model.

alpha : float

Power index of the beta model.

Other Parameters

fixed : a dict

A dictionary `{parameter_name: boolean}` of parameters to not be varied during fitting. True means the parameter is held fixed. Alternatively the `fixed` property of a parameter may be used.

tied : dict

A dictionary `{parameter_name: callable}` of parameters which are linked to some other parameter. The dictionary values are callables providing the linking relationship. Alternatively the `tied` property of a parameter may be used.

bounds : dict

A dictionary `{parameter_name: boolean}` of lower and upper bounds of parameters. Keys are parameter names. Values are a list of length 2 giving the desired range for the parameter. Alternatively the `min` and `max` properties of a parameter may be used.

eqcons : list

A list of functions of length n such that `eqcons[j](x0, *args) == 0.0` in a successfully optimized problem.

ineqcons : list

A list of functions of length n such that `ineqcons[j](x0, *args) >= 0.0` is a successfully optimized problem.

See also:

[Gaussian1D](#), [Box1D](#)

Notes

Model formula:

$$f(x) = A \left(1 + \frac{(x - x_0)^2}{\gamma^2} \right)^{-\alpha}$$

Attributes Summary

<code>alpha</code>	
<code>amplitude</code>	
<code>gamma</code>	
<code>param_names</code>	<code>list()</code> -> new empty list
<code>x_0</code>	

Methods Summary

<code>eval(x, amplitude, x_0, gamma, alpha)</code>	One dimensional Beta model function
<code>fit_deriv(x, amplitude, x_0, gamma, alpha)</code>	One dimensional Beta model derivative with respect to parameters

Attributes Documentation

alpha

amplitude

gamma

param_names = ['amplitude', 'x_0', 'gamma', 'alpha']

x_0

Methods Documentation

static eval (*x, amplitude, x_0, gamma, alpha*)

One dimensional Beta model function

static fit_deriv (*x, amplitude, x_0, gamma, alpha*)

One dimensional Beta model derivative with respect to parameters

Beta2D

class `astropy.modeling.functional_models.Beta2D` (*amplitude, x_0, y_0, gamma, alpha, **kwargs*)

Bases: `astropy.modeling.Fittable2DModel`

Two dimensional Beta model.

Parameters

amplitude : float

Amplitude of the model.

x_0 : float

x position of the maximum of the Beta model.

y_0 : float

y position of the maximum of the Beta model.

gamma : float

Core width of the Beta model.

alpha : float

Power index of the beta model.

Other Parameters**fixed** : a dict

A dictionary {parameter_name: boolean} of parameters to not be varied during fitting. True means the parameter is held fixed. Alternatively the `fixed` property of a parameter may be used.

tied : dict

A dictionary {parameter_name: callable} of parameters which are linked to some other parameter. The dictionary values are callables providing the linking relationship. Alternatively the `tied` property of a parameter may be used.

bounds : dict

A dictionary {parameter_name: boolean} of lower and upper bounds of parameters. Keys are parameter names. Values are a list of length 2 giving the desired range for the parameter. Alternatively the `min` and `max` properties of a parameter may be used.

eqcons : list

A list of functions of length n such that `eqcons[j](x0, *args) == 0.0` in a successfully optimized problem.

ineqcons : list

A list of functions of length n such that `ieqcons[j](x0, *args) >= 0.0` is a successfully optimized problem.

See also:[Gaussian2D](#), [Box2D](#)**Notes**

Model formula:

$$f(x, y) = A \left(1 + \frac{(x - x_0)^2 + (y - y_0)^2}{\gamma^2} \right)^{-\alpha}$$

Attributes Summary

alpha	
amplitude	
gamma	
param_names	list() -> new empty list
x_0	
y_0	

Methods Summary

<code>eval(x, y, amplitude, x_0, y_0, gamma, alpha)</code>	Two dimensional Beta model function
<code>fit_deriv(x, y, amplitude, x_0, y_0, gamma, ...)</code>	Two dimensional Beta model derivative with respect to parameters

Attributes Documentation

alpha

amplitude

gamma

param_names = ['amplitude', 'x_0', 'y_0', 'gamma', 'alpha']

x_0

y_0

Methods Documentation

static eval (*x*, *y*, *amplitude*, *x_0*, *y_0*, *gamma*, *alpha*)

Two dimensional Beta model function

static fit_deriv (*x*, *y*, *amplitude*, *x_0*, *y_0*, *gamma*, *alpha*)

Two dimensional Beta model derivative with respect to parameters

Box1D

class `astropy.modeling.functional_models.Box1D` (*amplitude*, *x_0*, *width*, ***kwargs*)

Bases: `astropy.modeling.Fittable1DModel`

One dimensional Box model.

Parameters

amplitude : float

Amplitude A

x_0 : float

Position of the center of the box function

width : float

Width of the box

Other Parameters

fixed : a dict

A dictionary {`parameter_name`: `boolean`} of parameters to not be varied during fitting. True means the parameter is held fixed. Alternatively the `fixed` property of a parameter may be used.

tied : dict

A dictionary {`parameter_name`: `callable`} of parameters which are linked to some other parameter. The dictionary values are callables providing the linking relationship. Alternatively the `tied` property of a parameter may be used.

bounds : dict

A dictionary {parameter_name: boolean} of lower and upper bounds of parameters. Keys are parameter names. Values are a list of length 2 giving the desired range for the parameter. Alternatively the `min` and `max` properties of a parameter may be used.

eqcons : list

A list of functions of length `n` such that `eqcons[j](x0, *args) == 0.0` in a successfully optimized problem.

ineqcons : list

A list of functions of length `n` such that `ineqcons[j](x0, *args) >= 0.0` is a successfully optimized problem.

See also:

`Box2D`, `TrapezoidDisk2D`

Notes

Model formula:

$$f(x) = \begin{cases} A & : x_0 - w/2 \geq x \geq x_0 + w/2 \\ 0 & : \text{else} \end{cases}$$

Attributes Summary

<code>amplitude</code>	
<code>param_names</code>	list() -> new empty list
<code>width</code>	
<code>x_0</code>	

Methods Summary

<code>eval(x, amplitude, x_0, width)</code>	One dimensional Box model function
<code>fit_deriv(x, amplitude, x_0, width)</code>	One dimensional Box model derivative with respect to parameters

Attributes Documentation

amplitude

`param_names = ['amplitude', 'x_0', 'width']`

width

x_0

Methods Documentation

static eval (*x*, *amplitude*, *x_0*, *width*)

One dimensional Box model function

classmethod fit_deriv (*x*, *amplitude*, *x_0*, *width*)

One dimensional Box model derivative with respect to parameters

Box2D

class `astropy.modeling.functional_models.Box2D` (*amplitude*, *x_0*, *y_0*, *x_width*, *y_width*,
***kwargs*)

Bases: `astropy.modeling.Fittable2DModel`

Two dimensional Box model.

Parameters

amplitude : float

Amplitude A

x_0 : float

x position of the center of the box function

x_width : float

Width in x direction of the box

y_0 : float

y position of the center of the box function

y_width : float

Width in y direction of the box

Other Parameters

fixed : a dict

A dictionary `{parameter_name: boolean}` of parameters to not be varied during fitting. True means the parameter is held fixed. Alternatively the `fixed` property of a parameter may be used.

tied : dict

A dictionary `{parameter_name: callable}` of parameters which are linked to some other parameter. The dictionary values are callables providing the linking relationship. Alternatively the `tied` property of a parameter may be used.

bounds : dict

A dictionary `{parameter_name: boolean}` of lower and upper bounds of parameters. Keys are parameter names. Values are a list of length 2 giving the desired range for the parameter. Alternatively the `min` and `max` properties of a parameter may be used.

eqcons : list

A list of functions of length *n* such that `eqcons[j](x0,*args) == 0.0` in a successfully optimized problem.

ineqcons : list

A list of functions of length n such that `ieqcons[j](x0, *args) >= 0.0` is a successfully optimized problem.

See also:

`Box1D`, `Gaussian2D`, `Beta2D`

Notes

Model formula:

$$f(x, y) = \begin{cases} A & : x_0 - w_x/2 \geq x \geq x_0 + w_x/2 \\ A & : y_0 - w_y/2 \geq y \geq y_0 + w_y/2 \\ 0 & : \text{else} \end{cases}$$

Attributes Summary

```
amplitude
param_names  list() -> new empty list
x_0
x_width
y_0
y_width
```

Methods Summary

```
eval(x, y, amplitude, x_0, y_0, x_width, y_width)  Two dimensional Box model function
```

Attributes Documentation

amplitude

```
param_names = ['amplitude', 'x_0', 'y_0', 'x_width', 'y_width']
```

x_0

x_width

y_0

y_width

Methods Documentation

static eval(*x*, *y*, *amplitude*, *x_0*, *y_0*, *x_width*, *y_width*)
Two dimensional Box model function

Const1D

class `astropy.modeling.functional_models.Const1D`(*amplitude*, ***kwargs*)
Bases: `astropy.modeling.Fittable1DModel`

One dimensional Constant model.

Parameters

amplitude : float

Value of the constant function

Other Parameters

fixed : a dict

A dictionary {*parameter_name*: *boolean*} of parameters to not be varied during fitting. True means the parameter is held fixed. Alternatively the *fixed* property of a parameter may be used.

tied : dict

A dictionary {*parameter_name*: *callable*} of parameters which are linked to some other parameter. The dictionary values are callables providing the linking relationship. Alternatively the *tied* property of a parameter may be used.

bounds : dict

A dictionary {*parameter_name*: *boolean*} of lower and upper bounds of parameters. Keys are parameter names. Values are a list of length 2 giving the desired range for the parameter. Alternatively the *min* and *max* properties of a parameter may be used.

eqcons : list

A list of functions of length *n* such that `eqcons[j](x0, *args) == 0.0` in a successfully optimized problem.

ineqcons : list

A list of functions of length *n* such that `ieqcons[j](x0, *args) >= 0.0` is a successfully optimized problem.

See also:

`Const2D`

Notes

Model formula:

$$f(x) = A$$

Attributes Summary

<code>amplitude</code>	
<code>linear</code>	<code>bool(x) -> bool</code>
<code>param_names</code>	<code>list() -> new empty list</code>

Methods Summary

<code>eval(x, amplitude)</code>	One dimensional Constant model function
<code>fit_deriv(x, amplitude)</code>	One dimensional Constant model derivative with respect to parameters

Attributes Documentation

amplitude

linear = True

param_names = ['amplitude']

Methods Documentation

static eval (*x*, *amplitude*)
One dimensional Constant model function

static fit_deriv (*x*, *amplitude*)
One dimensional Constant model derivative with respect to parameters

Const2D

class `astropy.modeling.functional_models.Const2D` (*amplitude*, ***kwargs*)

Bases: `astropy.modeling.Fittable2DModel`

Two dimensional Constant model.

Parameters

amplitude : float

Value of the constant function

Other Parameters

fixed : a dict

A dictionary `{parameter_name: boolean}` of parameters to not be varied during fitting. True means the parameter is held fixed. Alternatively the `fixed` property of a parameter may be used.

tied : dict

A dictionary `{parameter_name: callable}` of parameters which are linked to some other parameter. The dictionary values are callables providing the linking relationship. Alternatively the `tied` property of a parameter may be used.

bounds : dict

A dictionary {parameter_name: boolean} of lower and upper bounds of parameters. Keys are parameter names. Values are a list of length 2 giving the desired range for the parameter. Alternatively the `min` and `max` properties of a parameter may be used.

eqcons : list

A list of functions of length `n` such that `eqcons[j](x0,*args) == 0.0` in a successfully optimized problem.

ineqcons : list

A list of functions of length `n` such that `ieqcons[j](x0,*args) >= 0.0` is a successfully optimized problem.

See also:

`Const1D`

Notes

Model formula:

$$f(x,y) = A$$

Attributes Summary

<code>amplitude</code>	
<code>linear</code>	<code>bool(x) -> bool</code>
<code>param_names</code>	<code>list() -> new empty list</code>

Methods Summary

<code>eval(x, y, amplitude)</code>	Two dimensional Constant model function
------------------------------------	---

Attributes Documentation

amplitude

linear = True

param_names = ['amplitude']

Methods Documentation

static eval (*x*, *y*, *amplitude*)

Two dimensional Constant model function

Disk2D

class `astropy.modeling.functional_models.Disk2D` (*amplitude*, *x_0*, *y_0*, *R_0*, ***kwargs*)

Bases: `astropy.modeling.Fittable2DModel`

Two dimensional radial symmetric Disk model.

Parameters

amplitude : float

Value of the disk function

x_0 : float

x position center of the disk

y_0 : float

y position center of the disk

R_0 : float

Radius of the disk

Other Parameters

fixed : a dict

A dictionary {`parameter_name`: `boolean`} of parameters to not be varied during fitting. True means the parameter is held fixed. Alternatively the `fixed` property of a parameter may be used.

tied : dict

A dictionary {`parameter_name`: `callable`} of parameters which are linked to some other parameter. The dictionary values are callables providing the linking relationship. Alternatively the `tied` property of a parameter may be used.

bounds : dict

A dictionary {`parameter_name`: `boolean`} of lower and upper bounds of parameters. Keys are parameter names. Values are a list of length 2 giving the desired range for the parameter. Alternatively the `min` and `max` properties of a parameter may be used.

eqcons : list

A list of functions of length `n` such that `eqcons[j](x0, *args) == 0.0` in a successfully optimized problem.

ineqcons : list

A list of functions of length `n` such that `ieqcons[j](x0, *args) >= 0.0` is a successfully optimized problem.

See also:

`Box2D`, `TrapezoidDisk2D`

Notes

Model formula:

$$f(r) = \begin{cases} A & : r \leq R_0 \\ 0 & : r > R_0 \end{cases}$$

Attributes Summary

```
R_0
amplitude
param_names  list() -> new empty list
x_0
y_0
```

Methods Summary

```
eval(x, y, amplitude, x_0, y_0, R_0)  Two dimensional Disk model function
```

Attributes Documentation

R_0

amplitude

param_names = ['amplitude', 'x_0', 'y_0', 'R_0']

x_0

y_0

Methods Documentation

static eval (*x*, *y*, *amplitude*, *x_0*, *y_0*, *R_0*)
Two dimensional Disk model function

Gaussian1D

class `astropy.modeling.functional_models.Gaussian1D` (*amplitude*, *mean*, *stddev*, ***kwargs*)
Bases: `astropy.modeling.Fittable1DModel`

One dimensional Gaussian model.

Parameters

amplitude : float

Amplitude of the Gaussian.

mean : float

Mean of the Gaussian.

stddev : float

Standard deviation of the Gaussian.

Other Parameters

fixed : a dict

A dictionary `{parameter_name: boolean}` of parameters to not be varied during fitting. True means the parameter is held fixed. Alternatively the `fixed` property of a parameter may be used.

tied : dict

A dictionary `{parameter_name: callable}` of parameters which are linked to some other parameter. The dictionary values are callables providing the linking relationship. Alternatively the `tied` property of a parameter may be used.

bounds : dict

A dictionary `{parameter_name: boolean}` of lower and upper bounds of parameters. Keys are parameter names. Values are a list of length 2 giving the desired range for the parameter. Alternatively the `min` and `max` properties of a parameter may be used.

eqcons : list

A list of functions of length `n` such that `eqcons[j](x0, *args) == 0.0` in a successfully optimized problem.

ineqcons : list

A list of functions of length `n` such that `ineqcons[j](x0, *args) >= 0.0` is a successfully optimized problem.

See also:

[Gaussian2D](#), [Box1D](#), [Beta1D](#), [Lorentz1D](#)

Notes

Model formula:

$$f(x) = Ae^{-\frac{(x-x_0)^2}{2\sigma^2}}$$

Examples

```
>>> from astropy.modeling import models
>>> def tie_center(model):
...     mean = 50 * model.stddev
...     return mean
>>> tied_parameters = {'mean': tie_center}
```

Specify that 'mean' is a tied parameter in one of two ways:

```
>>> g1 = models.Gaussian1D(amplitude=10, mean=5, stddev=.3,
...                          tied=tied_parameters)
```

or

```
>>> g1 = models.Gaussian1D(amplitude=10, mean=5, stddev=.3)
>>> g1.mean.tied
False
>>> g1.mean.tied = tie_center
>>> g1.mean.tied
<function tie_center at 0x...>
```

Fixed parameters:

```
>>> g1 = models.Gaussian1D(amplitude=10, mean=5, stddev=.3,
...                          fixed={'stddev': True})
>>> g1.stddev.fixed
True
```

or

```
>>> g1 = models.Gaussian1D(amplitude=10, mean=5, stddev=.3)
>>> g1.stddev.fixed
False
>>> g1.stddev.fixed = True
>>> g1.stddev.fixed
True
```

Attributes Summary

```
amplitude
mean
param_names  list() -> new empty list
stddev
```

Methods Summary

```
eval(x, amplitude, mean, stddev)  Gaussian1D model function.
fit_deriv(x, amplitude, mean, stddev)  Gaussian1D model function derivatives.
```

Attributes Documentation

amplitude

mean

param_names = ['amplitude', 'mean', 'stddev']

stddev

Methods Documentation

static eval (*x, amplitude, mean, stddev*)
Gaussian1D model function.

static fit_deriv (*x, amplitude, mean, stddev*)
Gaussian1D model function derivatives.

Gaussian2D

```
class astropy.modeling.functional_models.Gaussian2D(amplitude, x_mean, y_mean,  
x_stddev=None, y_stddev=None,  
theta=0.0, cov_matrix=None,  
**kwargs)
```

Bases: `astropy.modeling.Fittable2DModel`

Two dimensional Gaussian model.

Parameters

amplitude : float

Amplitude of the Gaussian.

x_mean : float

Mean of the Gaussian in x.

y_mean : float

Mean of the Gaussian in y.

x_stddev : float

Standard deviation of the Gaussian in x. `x_stddev` and `y_stddev` must be specified unless a covariance matrix (`cov_matrix`) is input.

y_stddev : float

Standard deviation of the Gaussian in y. `x_stddev` and `y_stddev` must be specified unless a covariance matrix (`cov_matrix`) is input.

theta : float, optional

Rotation angle in radians. The rotation angle increases counterclockwise.

cov_matrix : ndarray, optional

A 2x2 covariance matrix. If specified, overrides the `x_stddev`, `y_stddev`, and `theta` specification.

Other Parameters

fixed : a dict

A dictionary `{parameter_name: boolean}` of parameters to not be varied during fitting. True means the parameter is held fixed. Alternatively the `fixed` property of a parameter may be used.

tied : dict

A dictionary `{parameter_name: callable}` of parameters which are linked to some other parameter. The dictionary values are callables providing the linking relationship. Alternatively the `tied` property of a parameter may be used.

bounds : dict

A dictionary `{parameter_name: boolean}` of lower and upper bounds of parameters. Keys are parameter names. Values are a list of length 2 giving the desired range for the parameter. Alternatively the `min` and `max` properties of a parameter may be used.

eqcons : list

A list of functions of length `n` such that `eqcons[j](x0, *args) == 0.0` in a successfully optimized problem.

ineqcons : list

A list of functions of length `n` such that `ineqcons[j](x0, *args) >= 0.0` is a successfully optimized problem.

See also:

`Gaussian1D`, `Box2D`, `Beta2D`

Notes

Model formula:

$$f(x, y) = Ae^{-a(x-x_0)^2 - b(x-x_0)(y-y_0) - c(y-y_0)^2}$$

Using the following definitions:

$$a = \left(\frac{\cos^2(\theta)}{2\sigma_x^2} + \frac{\sin^2(\theta)}{2\sigma_y^2} \right)$$

$$b = \left(\frac{\sin(2\theta)}{2\sigma_x^2} - \frac{\sin(2\theta)}{2\sigma_y^2} \right)$$

$$c = \left(\frac{\sin^2(\theta)}{2\sigma_x^2} + \frac{\cos^2(\theta)}{2\sigma_y^2} \right)$$

If using a `cov_matrix`, the model is of the form:

$$f(x, y) = Ae^{-0.5(\vec{x}-\vec{x}_0)^T \Sigma^{-1}(\vec{x}-\vec{x}_0)}$$

where $\vec{x} = [x, y]$, $\vec{x}_0 = [x_0, y_0]$, and Σ is the covariance matrix:

$$\Sigma = \begin{pmatrix} \sigma_x^2 & \rho\sigma_x\sigma_y \\ \rho\sigma_x\sigma_y & \sigma_y^2 \end{pmatrix}$$

ρ is the correlation between `x` and `y`, which should be between -1 and +1. Positive correlation corresponds to a `theta` in the range 0 to 90 degrees. Negative correlation corresponds to a `theta` in the range of 0 to -90 degrees.

See [R6] for more details about the 2D Gaussian function.

References

[R6]

Attributes Summary

<code>amplitude</code>	
<code>param_names</code>	<code>list()</code> -> new empty list
<code>theta</code>	
<code>x_mean</code>	
<code>x_stddev</code>	
<code>y_mean</code>	
<code>y_stddev</code>	

Methods Summary

<code>eval(x, y, amplitude, x_mean, y_mean, ...)</code>	Two dimensional Gaussian function
<code>fit_deriv(x, y, amplitude, x_mean, y_mean, ...)</code>	Two dimensional Gaussian function derivative with respect to parameters

Attributes Documentation

amplitude

param_names = ['amplitude', 'x_mean', 'y_mean', 'x_stddev', 'y_stddev', 'theta']

theta

x_mean

x_stddev

y_mean

y_stddev

Methods Documentation

static eval (*x, y, amplitude, x_mean, y_mean, x_stddev, y_stddev, theta*)

Two dimensional Gaussian function

static fit_deriv (*x, y, amplitude, x_mean, y_mean, x_stddev, y_stddev, theta*)

Two dimensional Gaussian function derivative with respect to parameters

GaussianAbsorption1D

class `astropy.modeling.functional_models.GaussianAbsorption1D` (*amplitude, mean, std-dev, **kwargs*)

Bases: `astropy.modeling.Fittable1DModel`

One dimensional Gaussian absorption line model.

Parameters

amplitude : float

Amplitude of the gaussian absorption.

mean : float

Mean of the gaussian.

stddev : float

Standard deviation of the gaussian.

Other Parameters

fixed : a dict

A dictionary `{parameter_name: boolean}` of parameters to not be varied during fitting. True means the parameter is held fixed. Alternatively the `fixed` property of a parameter may be used.

tied : dict

A dictionary `{parameter_name: callable}` of parameters which are linked to some other parameter. The dictionary values are callables providing the linking relationship. Alternatively the `tied` property of a parameter may be used.

bounds : dict

A dictionary `{parameter_name: boolean}` of lower and upper bounds of parameters. Keys are parameter names. Values are a list of length 2 giving the desired range for the parameter. Alternatively the `min` and `max` properties of a parameter may be used.

eqcons : list

A list of functions of length `n` such that `eqcons[j](x0, *args) == 0.0` in a successfully optimized problem.

ineqcons : list

A list of functions of length `n` such that `ineqcons[j](x0, *args) >= 0.0` is a successfully optimized problem.

See also:

`Gaussian1D`

Notes

Model formula:

$$f(x) = 1 - Ae^{-\frac{(x-x_0)^2}{2\sigma^2}}$$

Attributes Summary

<code>amplitude</code>	
<code>mean</code>	
<code>param_names</code>	list() -> new empty list
<code>stddev</code>	

Methods Summary

<code>eval(x, amplitude, mean, stddev)</code>	GaussianAbsorption1D model function.
<code>fit_deriv(x, amplitude, mean, stddev)</code>	GaussianAbsorption1D model function derivatives.

Attributes Documentation

amplitude

mean

param_names = ['amplitude', 'mean', 'stddev']

stddev

Methods Documentation

static eval (*x, amplitude, mean, stddev*)
GaussianAbsorption1D model function.

static fit_deriv (*x, amplitude, mean, stddev*)
GaussianAbsorption1D model function derivatives.

Linear1D

class `astropy.modeling.functional_models.Linear1D` (*slope, intercept, **kwargs*)
Bases: `astropy.modeling.Fittable1DModel`

One dimensional Line model.

Parameters

slope : float

Slope of the straight line

intercept : float

Intercept of the straight line

Other Parameters

fixed : a dict

A dictionary `{parameter_name: boolean}` of parameters to not be varied during fitting. True means the parameter is held fixed. Alternatively the `fixed` property of a parameter may be used.

tied : dict

A dictionary `{parameter_name: callable}` of parameters which are linked to some other parameter. The dictionary values are callables providing the linking relationship. Alternatively the `tied` property of a parameter may be used.

bounds : dict

A dictionary `{parameter_name: boolean}` of lower and upper bounds of parameters. Keys are parameter names. Values are a list of length 2 giving the desired range for the parameter. Alternatively the `min` and `max` properties of a parameter may be used.

eqcons : list

A list of functions of length `n` such that `eqcons[j](x0, *args) == 0.0` in a successfully optimized problem.

ineqcons : list

A list of functions of length `n` such that `ineqcons[j](x0, *args) >= 0.0` is a successfully optimized problem.

See also:

`Const1D`

Notes

Model formula:

$$f(x) = ax + b$$

Attributes Summary

<code>intercept</code>	
<code>linear</code>	<code>bool(x) -> bool</code>
<code>param_names</code>	<code>list() -> new empty list</code>
<code>slope</code>	

Methods Summary

<code>eval(x, slope, intercept)</code>	One dimensional Line model function
<code>fit_deriv(x, slope, intercept)</code>	One dimensional Line model derivative with respect to parameters

Attributes Documentation

intercept

```
linear = True
```

```
param_names = ['slope', 'intercept']
```

```
slope
```

Methods Documentation

static eval (*x*, *slope*, *intercept*)

One dimensional Line model function

static fit_deriv (*x*, *slope*, *intercept*)

One dimensional Line model derivative with respect to parameters

Lorentz1D

class `astropy.modeling.functional_models.Lorentz1D` (*amplitude*, *x_0*, *fwhm*, ***kwargs*)

Bases: `astropy.modeling.Fittable1DModel`

One dimensional Lorentzian model.

Parameters

amplitude : float

Peak value

x_0 : float

Position of the peak

fwhm : float

Full width at half maximum

Other Parameters

fixed : a dict

A dictionary {`parameter_name`: `boolean`} of parameters to not be varied during fitting. True means the parameter is held fixed. Alternatively the `fixed` property of a parameter may be used.

tied : dict

A dictionary {`parameter_name`: `callable`} of parameters which are linked to some other parameter. The dictionary values are callables providing the linking relationship. Alternatively the `tied` property of a parameter may be used.

bounds : dict

A dictionary {`parameter_name`: `boolean`} of lower and upper bounds of parameters. Keys are parameter names. Values are a list of length 2 giving the desired range for the parameter. Alternatively the `min` and `max` properties of a parameter may be used.

eqcons : list

A list of functions of length `n` such that `eqcons[j](x0, *args) == 0.0` in a successfully optimized problem.

ineqcons : list

A list of functions of length n such that `ineqcons[j](x0, *args) >= 0.0` is a successfully optimized problem.

See also:

`Gaussian1D`, `Box1D`, `MexicanHat1D`

Notes

Model formula:

$$f(x) = \frac{A\gamma^2}{\gamma^2 + (x - x_0)^2}$$

Attributes Summary

```
amplitude
fwhm
param_names  list() -> new empty list
x_0
```

Methods Summary

<code>eval(x, amplitude, x_0, fwhm)</code>	One dimensional Lorentzian model function
<code>fit_deriv(x, amplitude, x_0, fwhm)</code>	One dimensional Lorentzian model derivative wiht respect to parameters

Attributes Documentation

amplitude

fwhm

param_names = ['amplitude', 'x_0', 'fwhm']

x_0

Methods Documentation

static eval (*x, amplitude, x_0, fwhm*)
One dimensional Lorentzian model function

static fit_deriv (*x, amplitude, x_0, fwhm*)
One dimensional Lorentzian model derivative wiht respect to parameters

MexicanHat1D

```
class astropy.modeling.functional_models.MexicanHat1D(amplitude, x_0, sigma,
**kwargs)
```

Bases: `astropy.modeling.Fittable1DModel`

One dimensional Mexican Hat model.

Parameters

amplitude : float

Amplitude

x_0 : float

Position of the peak

sigma : float

Width of the Mexican hat

Other Parameters

fixed : a dict

A dictionary {parameter_name: boolean} of parameters to not be varied during fitting. True means the parameter is held fixed. Alternatively the `fixed` property of a parameter may be used.

tied : dict

A dictionary {parameter_name: callable} of parameters which are linked to some other parameter. The dictionary values are callables providing the linking relationship. Alternatively the `tied` property of a parameter may be used.

bounds : dict

A dictionary {parameter_name: boolean} of lower and upper bounds of parameters. Keys are parameter names. Values are a list of length 2 giving the desired range for the parameter. Alternatively the `min` and `max` properties of a parameter may be used.

eqcons : list

A list of functions of length `n` such that `eqcons[j](x0, *args) == 0.0` in a successfully optimized problem.

ineqcons : list

A list of functions of length `n` such that `ineqcons[j](x0, *args) >= 0.0` is a successfully optimized problem.

See also:

`MexicanHat2D`, `Box1D`, `Gaussian1D`, `Trapezoid1D`

Notes

Model formula:

$$f(x) = A \left(1 - \frac{(x - x_0)^2}{\sigma^2} \right) e^{-\frac{(x - x_0)^2}{2\sigma^2}}$$

Attributes Summary

```

amplitude
param_names  list() -> new empty list
sigma
x_0

```

Methods Summary

```

eval(x, amplitude, x_0, sigma)  One dimensional Mexican Hat model function

```

Attributes Documentation

amplitude

param_names = ['amplitude', 'x_0', 'sigma']

sigma

x_0

Methods Documentation

static eval (*x, amplitude, x_0, sigma*)
One dimensional Mexican Hat model function

MexicanHat2D

class astropy.modeling.functional_models.**MexicanHat2D** (*amplitude, x_0, y_0, sigma,*
***kwargs*)

Bases: astropy.modeling.Fittable2DModel

Two dimensional symmetric Mexican Hat model.

Parameters

amplitude : float

Amplitude

x_0 : float

x position of the peak

y_0 : float

y position of the peak

sigma : float

Width of the Mexican hat

Other Parameters**fixed** : a dict

A dictionary {parameter_name: boolean} of parameters to not be varied during fitting. True means the parameter is held fixed. Alternatively the `fixed` property of a parameter may be used.

tied : dict

A dictionary {parameter_name: callable} of parameters which are linked to some other parameter. The dictionary values are callables providing the linking relationship. Alternatively the `tied` property of a parameter may be used.

bounds : dict

A dictionary {parameter_name: boolean} of lower and upper bounds of parameters. Keys are parameter names. Values are a list of length 2 giving the desired range for the parameter. Alternatively the `min` and `max` properties of a parameter may be used.

eqcons : list

A list of functions of length n such that `eqcons[j](x0, *args) == 0.0` in a successfully optimized problem.

ineqcons : list

A list of functions of length n such that `ieqcons[j](x0, *args) >= 0.0` is a successfully optimized problem.

See also:

`MexicanHat1D`, `Gaussian2D`

Notes

Model formula:

$$f(x, y) = A \left(1 - \frac{(x - x_0)^2 + (y - y_0)^2}{\sigma^2} \right) e^{-\frac{(x - x_0)^2 + (y - y_0)^2}{2\sigma^2}}$$

Attributes Summary

<code>amplitude</code>	
<code>param_names</code>	list() -> new empty list
<code>sigma</code>	
<code>x_0</code>	
<code>y_0</code>	

Methods Summary

<code>eval(x, y, amplitude, x_0, y_0, sigma)</code>	Two dimensional Mexican Hat model function
---	--

Attributes Documentation

amplitude

```
param_names = ['amplitude', 'x_0', 'y_0', 'sigma']
```

sigma

x_0

y_0

Methods Documentation

static eval (*x*, *y*, *amplitude*, *x_0*, *y_0*, *sigma*)

Two dimensional Mexican Hat model function

Redshift

class `astropy.modeling.functional_models.Redshift` (*z*, ***kwargs*)

Bases: `astropy.modeling.Fittable1DModel`

One dimensional redshift model.

Parameters

z : float or a list of floats

Redshift value(s).

Notes

Model formula:

$$\lambda_{obs} = (1 + z)\lambda_{rest}$$

Attributes Summary

<code>param_names</code>	list() -> new empty list
<code>z</code>	redshift

Methods Summary

<code>eval(x, z)</code>	One dimensional Redshift model function
<code>fit_deriv(x, z)</code>	One dimensional Redshift model derivative

Continued on next page

Table 12.57 – continued from previous page

<code>inverse()</code>	Inverse Redshift model
------------------------	------------------------

Attributes Documentation

`param_names = ['z']`

z
redshift

Methods Documentation

static eval (*x*, *z*)
One dimensional Redshift model function

static fit_deriv (*x*, *z*)
One dimensional Redshift model derivative

inverse ()
Inverse Redshift model

Ring2D

class `astropy.modeling.functional_models.Ring2D` (*amplitude*, *x_0*, *y_0*, *r_in*, *width=None*,
r_out=None, ***kwargs*)

Bases: `astropy.modeling.Fittable2DModel`

Two dimensional radial symmetric Ring model.

Parameters

amplitude : float

Value of the disk function

x_0 : float

x position center of the disk

y_0 : float

y position center of the disk

r_in : float

Inner radius of the ring

width : float

Width of the ring.

r_out : float

Outer Radius of the ring. Can be specified instead of width.

See also:

`Disk2D`, `TrapezoidDisk2D`

Notes

Model formula:

$$f(r) = \begin{cases} A & : r_{in} \leq r \leq r_{out} \\ 0 & : \text{else} \end{cases}$$

Where $r_{out} = r_{in} + r_{width}$.

Attributes Summary

```

amplitude
param_names  list() -> new empty list
r_in
width
x_0
y_0

```

Methods Summary

```

eval(x, y, amplitude, x_0, y_0, r_in, width)  Two dimensional Ring model function.

```

Attributes Documentation

amplitude

param_names = ['amplitude', 'x_0', 'y_0', 'r_in', 'width']

r_in

width

x_0

y_0

Methods Documentation

static eval (*x*, *y*, *amplitude*, *x_0*, *y_0*, *r_in*, *width*)

Two dimensional Ring model function.

Scale

class `astropy.modeling.functional_models.Scale` (*factors*, ***kwargs*)

Bases: `astropy.modeling.Model`

Multiply a model by a factor.

Parameters

factors : float or a list of floats

scale for a coordinate

Attributes Summary

<code>factors</code>	
<code>linear</code>	<code>bool(x) -> bool</code>
<code>param_names</code>	<code>list() -> new empty list</code>

Methods Summary

<code>__call__</code> (<i>*inputs</i> , <i>**kwargs</i>)	Transforms data using this model.
<code>inverse</code> ()	

Attributes Documentation

factors

linear = True

param_names = ['factors']

Methods Documentation

`__call__` (**inputs*, ***kwargs*)
Transforms data using this model.

Parameters

x : array like or a number

input

inverse ()

Shift

class `astropy.modeling.functional_models.Shift` (*offsets*, ***kwargs*)

Bases: `astropy.modeling.Model`

Shift a coordinate.

Parameters

offsets : float or a list of floats

offsets to be applied to a coordinate if a list - each value in the list is an offset to be applied to a column in the input coordinate array

Attributes Summary

<code>offsets</code>	
<code>param_names</code>	list() -> new empty list

Methods Summary

<code>__call__(*inputs, **kwargs)</code>	Transforms data using this model.
<code>inverse()</code>	

Attributes Documentation

offsets

param_names = ['offsets']

Methods Documentation

`__call__(*inputs, **kwargs)`
Transforms data using this model.

Parameters

x : array like or a number

input

inverse()

Sine1D

class `astropy.modeling.functional_models.Sine1D(amplitude, frequency, **kwargs)`
Bases: `astropy.modeling.Fittable1DModel`

One dimensional Sine model.

Parameters

amplitude : float

Oscillation amplitude

frequency : float

Oscillation frequency

Other Parameters

fixed : a dict

A dictionary `{parameter_name: boolean}` of parameters to not be varied during fitting. True means the parameter is held fixed. Alternatively the `fixed` property of a parameter may be used.

tied : dict

A dictionary `{parameter_name: callable}` of parameters which are linked to some other parameter. The dictionary values are callables providing the linking relationship. Alternatively the `tied` property of a parameter may be used.

bounds : dict

A dictionary `{parameter_name: boolean}` of lower and upper bounds of parameters. Keys are parameter names. Values are a list of length 2 giving the desired range for the parameter. Alternatively the `min` and `max` properties of a parameter may be used.

eqcons : list

A list of functions of length `n` such that `eqcons[j](x0, *args) == 0.0` in a successfully optimized problem.

ineqcons : list

A list of functions of length `n` such that `ineqcons[j](x0, *args) >= 0.0` is a successfully optimized problem.

See also:

`Const1D`, `Linear1D`

Notes

Model formula:

$$f(x) = A \sin(2\pi f x)$$

Attributes Summary

<code>amplitude</code>	
<code>frequency</code>	
<code>param_names</code>	<code>list()</code> -> new empty list

Methods Summary

<code>eval(x, amplitude, frequency)</code>	One dimensional Sine model function
<code>fit_deriv(x, amplitude, frequency)</code>	One dimensional Sine model derivative

Attributes Documentation

amplitude

frequency

param_names = ['amplitude', 'frequency']

Methods Documentation

static eval (*x*, *amplitude*, *frequency*)

One dimensional Sine model function

static fit_deriv (*x*, *amplitude*, *frequency*)

One dimensional Sine model derivative

Trapezoid1D

class `astropy.modeling.functional_models.Trapezoid1D` (*amplitude*, *x_0*, *width*, *slope*,
***kwargs*)

Bases: `astropy.modeling.Fittable1DModel`

One dimensional Trapezoid model.

Parameters

amplitude : float

Amplitude of the trapezoid

x_0 : float

Center position of the trapezoid

width : float

Width of the constant part of the trapezoid.

slope : float

Slope of the tails of the trapezoid

Other Parameters

fixed : a dict

A dictionary {`parameter_name`: `boolean`} of parameters to not be varied during fitting. True means the parameter is held fixed. Alternatively the `fixed` property of a parameter may be used.

tied : dict

A dictionary {`parameter_name`: `callable`} of parameters which are linked to some other parameter. The dictionary values are callables providing the linking relationship. Alternatively the `tied` property of a parameter may be used.

bounds : dict

A dictionary `{parameter_name: boolean}` of lower and upper bounds of parameters. Keys are parameter names. Values are a list of length 2 giving the desired range for the parameter. Alternatively the `min` and `max` properties of a parameter may be used.

eqcons : list

A list of functions of length `n` such that `eqcons[j](x0,*args) == 0.0` in a successfully optimized problem.

ineqcons : list

A list of functions of length `n` such that `ineqcons[j](x0,*args) >= 0.0` is a successfully optimized problem.

See also:

`Box1D`, `Gaussian1D`, `Beta1D`

Attributes Summary

<code>amplitude</code>	
<code>param_names</code>	<code>list()</code> -> new empty list
<code>slope</code>	
<code>width</code>	
<code>x_0</code>	

Methods Summary

<code>eval(x, amplitude, x_0, width, slope)</code>	One dimensional Trapezoid model function
--	--

Attributes Documentation

amplitude

param_names = ['amplitude', 'x_0', 'width', 'slope']

slope

width

x_0

Methods Documentation

static eval (*x, amplitude, x_0, width, slope*)
One dimensional Trapezoid model function

TrapezoidDisk2D

class `astropy.modeling.functional_models.TrapezoidDisk2D` (*amplitude*, *x_0*, *y_0*, *R_0*, *slope*, ***kwargs*)

Bases: `astropy.modeling.Fittable2DModel`

Two dimensional circular Trapezoid model.

Parameters

amplitude : float

Amplitude of the trapezoid

x_0 : float

x position of the center of the trapezoid

y_0 : float

y position of the center of the trapezoid

R_0 : float

Radius of the constant part of the trapezoid.

slope : float

Slope of the tails of the trapezoid in x direction.

Other Parameters

fixed : a dict

A dictionary `{parameter_name: boolean}` of parameters to not be varied during fitting. True means the parameter is held fixed. Alternatively the `fixed` property of a parameter may be used.

tied : dict

A dictionary `{parameter_name: callable}` of parameters which are linked to some other parameter. The dictionary values are callables providing the linking relationship. Alternatively the `tied` property of a parameter may be used.

bounds : dict

A dictionary `{parameter_name: boolean}` of lower and upper bounds of parameters. Keys are parameter names. Values are a list of length 2 giving the desired range for the parameter. Alternatively the `min` and `max` properties of a parameter may be used.

eqcons : list

A list of functions of length `n` such that `eqcons[j](x0,*args) == 0.0` in a successfully optimized problem.

ineqcons : list

A list of functions of length `n` such that `ineqcons[j](x0,*args) >= 0.0` is a successfully optimized problem.

See also:

`Disk2D`, `Box2D`

Attributes Summary

```

R_0
amplitude
param_names  list() -> new empty list
slope
x_0
y_0

```

Methods Summary

```

eval(x, y, amplitude, x_0, y_0, R_0, slope)  Two dimensional Trapezoid Disk model function

```

Attributes Documentation

R_0

amplitude

param_names = ['amplitude', 'x_0', 'y_0', 'R_0', 'slope']

slope

x_0

y_0

Methods Documentation

static eval (*x*, *y*, *amplitude*, *x_0*, *y_0*, *R_0*, *slope*)
 Two dimensional Trapezoid Disk model function

Class Inheritance Diagram

12.4.4 astropy.modeling.optimizers Module

Optimization algorithms used in *fitting*.

Classes

<code>Optimization(opt_method)</code>	Base class for optimizers.
<code>SLSQP()</code>	Sequential Least Squares Programming optimization algorithm.
<code>Simplex()</code>	Neald-Mead (downhill simplex) algorithm [1].

Optimization

class `astropy.modeling.optimizers.Optimization` (*opt_method*)

Bases: `object`

Base class for optimizers.

Parameters

opt_method : callable

Implements optimization method

Notes

The base Optimizer does not support any constraints by default; individual optimizers should explicitly set this list to the specific constraints it supports.

Attributes Summary

<code>acc</code>	Requested accuracy
<code>eps</code>	Step for the forward difference approximation of the Jacobian
<code>maxiter</code>	Maximum number of iterations
<code>opt_method</code>	
<code>supported_constraints</code>	<code>list()</code> -> new empty list

Methods Summary

`__call__()`

Attributes Documentation

acc

Requested accuracy

eps

Step for the forward difference approximation of the Jacobian

maxiter

Maximum number of iterations

opt_method

supported_constraints = []

Methods Documentation

`__call__()`

SLSQP

class `astropy.modeling.optimizers.SLSQP`

Bases: `astropy.modeling.optimizers.Optimization`

Sequential Least Squares Programming optimization algorithm.

The algorithm is described in [R7]. It supports tied and fixed parameters, as well as bounded constraints. Uses `scipy.optimize.fmin_slsqp`.

References

[R7]

Attributes Summary

`supported_constraints` list() -> new empty list

Methods Summary

`__call__(objfunc, initval, fargs, **kwargs)` Run the solver.

Attributes Documentation

`supported_constraints = [u'bounds', u'eqcons', u'ineqcons', u'fixed', u'tied']`

Methods Documentation

`__call__(objfunc, initval, fargs, **kwargs)`

Run the solver.

Parameters

objfunc : callable

objection function

initval : iterable

initial guess for the parameter values

fargs : tuple

other arguments to be passed to the statistic function

kwargs : dict

other keyword arguments to be passed to the solver

Simplex

class `astropy.modeling.optimizers.Simplex`

Bases: `astropy.modeling.optimizers.Optimization`

Neald-Mead (downhill simplex) algorithm [1].

This algorithm only uses function values, not derivatives. Uses `scipy.optimize.fmin`.

Attributes Summary

`supported_constraints` list() -> new empty list

Methods Summary

`__call__`(`objfunc`, `initval`, `fargs`, `**kwargs`) Run the solver.

Attributes Documentation

`supported_constraints` = ['u'bounds', u'fixed', u'tied']

Methods Documentation

`__call__`(`objfunc`, `initval`, `fargs`, `**kwargs`)
Run the solver.

Parameters

objfunc : callable

objection function

initval : iterable

initial guess for the parameter values

fargs : tuple

other arguments to be passed to the statistic function

kwargs : dict

other keyword arguments to be passed to the solver

Class Inheritance Diagram

12.4.5 `astropy.modeling.powerlaws` Module

Power law model variants

Classes

<code>BrokenPowerLaw1D(amplitude, x_break, ...)</code>	One dimensional power law model with a break.
<code>ExponentialCutoffPowerLaw1D(amplitude, x_0, ...)</code>	One dimensional power law model with an exponential cutoff.
<code>LogParabola1D(amplitude, x_0, alpha, beta, ...)</code>	One dimensional log parabola model (sometimes called curved power law).
<code>PowerLaw1D(amplitude, x_0, alpha, **constraints)</code>	One dimensional power law model.

BrokenPowerLaw1D

class `astropy.modeling.powerlaws.BrokenPowerLaw1D` (*amplitude, x_break, alpha_1, alpha_2, **constraints*)

Bases: `astropy.modeling.Fittable1DModel`

One dimensional power law model with a break.

Parameters

amplitude : float

Model amplitude at the break point

x_break : float

Break point

alpha_1 : float

Power law index for $x < x_{\text{break}}$

alpha_2 : float

Power law index for $x > x_{\text{break}}$

See also:

`PowerLaw1D`, `ExponentialCutoffPowerLaw1D`, `LogParabola1D`

Notes

Model formula (with A for amplitude and α_1 for alpha_1 and α_2 for alpha_2):

$$f(x) = \begin{cases} A(x/x_{\text{break}})^{-\alpha_1} & : x < x_{\text{break}} \\ A(x/x_{\text{break}})^{-\alpha_2} & : x > x_{\text{break}} \end{cases}$$

Attributes Summary

```
alpha_1
alpha_2
amplitude
param_names  list() -> new empty list
x_break
```

Methods Summary

<code>eval(x, amplitude, x_break, alpha_1, alpha_2)</code>	One dimensional broken power law model function
--	---

Continued on next page

Table 12.79 – continued from previous page

<code>fit_deriv(x, amplitude, x_break, alpha_1, ...)</code>	One dimensional broken power law derivative with respect to parameters
---	--

Attributes Documentation**alpha_1****alpha_2****amplitude****param_names** = ['amplitude', 'x_break', 'alpha_1', 'alpha_2']**x_break****Methods Documentation**

static eval (*x, amplitude, x_break, alpha_1, alpha_2*)
 One dimensional broken power law model function

static fit_deriv (*x, amplitude, x_break, alpha_1, alpha_2*)
 One dimensional broken power law derivative with respect to parameters

ExponentialCutoffPowerLaw1D

class `astropy.modeling.powerlaws.ExponentialCutoffPowerLaw1D` (*amplitude, x_0, alpha, x_cutoff, **constraints*)

Bases: `astropy.modeling.Fittable1DModel`

One dimensional power law model with an exponential cutoff.

Parameters

amplitude : float
 Model amplitude

x_0 : float
 Reference point

alpha : float
 Power law index

x_cutoff : float
 Cutoff point

See also:

`PowerLaw1D`, `BrokenPowerLaw1D`, `LogParabola1D`

Notes

Model formula (with A for amplitude and α for alpha):

$$f(x) = A(x/x_0)^{-\alpha} \exp(-x/x_{cutoff})$$

Attributes Summary

```
alpha
amplitude
param_names  list() -> new empty list
x_0
x_cutoff
```

Methods Summary

<code>eval(x, amplitude, x_0, alpha, x_cutoff)</code>	One dimensional exponential cutoff power law model function
<code>fit_deriv(x, amplitude, x_0, alpha, x_cutoff)</code>	One dimensional exponential cutoff power law derivative with respect to parameters

Attributes Documentation

alpha

amplitude

param_names = ['amplitude', 'x_0', 'alpha', 'x_cutoff']

x_0

x_cutoff

Methods Documentation

static eval (*x, amplitude, x_0, alpha, x_cutoff*)

One dimensional exponential cutoff power law model function

static fit_deriv (*x, amplitude, x_0, alpha, x_cutoff*)

One dimensional exponential cutoff power law derivative with respect to parameters

LogParabola1D

class `astropy.modeling.powerlaws.LogParabola1D` (*amplitude, x_0, alpha, beta, **constraints*)
Bases: `astropy.modeling.Fittable1DModel`

One dimensional log parabola model (sometimes called curved power law).

Parameters

amplitude : float

Model amplitude

x_0 : float

Reference point

alpha : float

Power law index

beta : float

Power law curvature

See also:

[PowerLaw1D](#), [BrokenPowerLaw1D](#), [ExponentialCutoffPowerLaw1D](#)

Notes

Model formula (with A for amplitude and α for alpha and β for beta):

$$f(x) = A \left(\frac{x}{x_0} \right)^{-\alpha - \beta \log \left(\frac{x}{x_0} \right)}$$

Attributes Summary

```
alpha
amplitude
beta
param_names  list() -> new empty list
x_0
```

Methods Summary

<code>eval(x, amplitude, x_0, alpha, beta)</code>	One dimensional log parabola model function
<code>fit_deriv(x, amplitude, x_0, alpha, beta)</code>	One dimensional log parabola derivative with respect to parameters

Attributes Documentation

alpha

amplitude

beta

```
param_names = ['amplitude', 'x_0', 'alpha', 'beta']
```

```
x_0
```

Methods Documentation

```
static eval(x, amplitude, x_0, alpha, beta)
```

One dimensional log parabola model function

```
static fit_deriv(x, amplitude, x_0, alpha, beta)
```

One dimensional log parabola derivative with respect to parameters

PowerLaw1D

```
class astropy.modeling.powerlaws.PowerLaw1D(amplitude, x_0, alpha, **constraints)
```

Bases: `astropy.modeling.Fittable1DModel`

One dimensional power law model.

Parameters

amplitude : float

Model amplitude at the reference point

x_0 : float

Reference point

alpha : float

Power law index

Other Parameters

fixed : a dict

A dictionary {parameter_name: boolean} of parameters to not be varied during fitting. True means the parameter is held fixed. Alternatively the `fixed` property of a parameter may be used.

tied : dict

A dictionary {parameter_name: callable} of parameters which are linked to some other parameter. The dictionary values are callables providing the linking relationship. Alternatively the `tied` property of a parameter may be used.

bounds : dict

A dictionary {parameter_name: boolean} of lower and upper bounds of parameters. Keys are parameter names. Values are a list of length 2 giving the desired range for the parameter. Alternatively the `min` and `max` properties of a parameter may be used.

eqcons : list

A list of functions of length `n` such that `eqcons[j](x0, *args) == 0.0` in a successfully optimized problem.

ineqcons : list

A list of functions of length `n` such that `ieqcons[j](x0, *args) >= 0.0` is a successfully optimized problem.

See also:

`BrokenPowerLaw1D`, `ExponentialCutoffPowerLaw1D`, `LogParabolalD`

Notes

Model formula (with A for amplitude and α for alpha):

$$f(x) = A(x/x_0)^{-\alpha}$$

Attributes Summary

<code>alpha</code>	
<code>amplitude</code>	
<code>param_names</code>	list() -> new empty list
<code>x_0</code>	

Methods Summary

<code>eval(x, amplitude, x_0, alpha)</code>	One dimensional power law model function
<code>fit_deriv(x, amplitude, x_0, alpha)</code>	One dimensional power law derivative with respect to parameters

Attributes Documentation

alpha

amplitude

param_names = ['amplitude', 'x_0', 'alpha']

x_0

Methods Documentation

static eval (*x, amplitude, x_0, alpha*)
One dimensional power law model function

static fit_deriv (*x, amplitude, x_0, alpha*)
One dimensional power law derivative with respect to parameters

Class Inheritance Diagram

12.4.6 astropy.modeling.polynomial Module

This module contains predefined polynomial models.

Classes

<code>Chebyshev1D(degree[, domain, window, ...])</code>	1D Chebyshev polynomial of the 1st kind.
<code>Chebyshev2D(x_degree, y_degree[, x_domain, ...])</code>	2D Chebyshev polynomial of the 1st kind.
<code>InverseSIP(ap_order, bp_order[, ap_coeff, ...])</code>	Inverse Simple Imaging Polynomial
<code>Legendre1D(degree[, domain, window, ...])</code>	1D Legendre polynomial.
<code>Legendre2D(x_degree, y_degree[, x_domain, ...])</code>	Legendre 2D polynomial.
<code>Polynomial1D(degree[, domain, window, ...])</code>	1D Polynomial model.
<code>Polynomial2D(degree[, x_domain, y_domain, ...])</code>	2D Polynomial model.
<code>SIP(crpix, a_order, b_order[, a_coeff, ...])</code>	Simple Imaging Polynomial (SIP) model.
<code>OrthoPolynomialBase(x_degree, y_degree[, ...])</code>	This is a base class for the 2D Chebyshev and Legendre models.
<code>PolynomialModel(degree[, n_inputs, ...])</code>	Base class for polynomial models.

Chebyshev1D

```
class astropy.modeling.polynomial.Chebyshev1D(degree, domain=None, window=[-1, 1],
                                             n_models=None, model_set_axis=None,
                                             **params)
```

Bases: `astropy.modeling.polynomial.PolynomialModel`

1D Chebyshev polynomial of the 1st kind.

Parameters

degree : int

degree of the series

domain : list or None

window : list or None

If None, it is set to [-1,1] Fitters will remap the domain to this window

param_dim : int

number of parameter sets

****params** : dict

keyword : value pairs, representing parameter_name: value

Other Parameters

fixed : a dict

A dictionary {parameter_name: boolean} of parameters to not be varied during fitting. True means the parameter is held fixed. Alternatively the `fixed` property of a parameter may be used.

tied : dict

A dictionary {parameter_name: callable} of parameters which are linked to some other parameter. The dictionary values are callables providing the linking relationship. Alternatively the `tied` property of a parameter may be used.

bounds : dict

A dictionary {parameter_name: boolean} of lower and upper bounds of parameters. Keys are parameter names. Values are a list of length 2 giving the desired range for the parameter. Alternatively the `min` and `max` properties of a parameter may be used.

eqcons : list

A list of functions of length `n` such that `eqcons[j](x0, *args) == 0.0` in a successfully optimized problem.

ineqcons : list

A list of functions of length `n` such that `ieqcons[j](x0, *args) >= 0.0` is a successfully optimized problem.

Methods Summary

<code>__call__(*inputs, **kwargs)</code>	Transforms data using this model.
<code>clenshaw(x, coeff)</code>	Evaluates the polynomial using Clenshaw's algorithm.
<code>fit_deriv(x, *params)</code>	Computes the Vandermonde matrix.

Methods Documentation

`__call__(*inputs, **kwargs)`
Transforms data using this model.

Parameters

x : scalar, list or array

input

`clenshaw(x, coeff)`
Evaluates the polynomial using Clenshaw's algorithm.

`fit_deriv(x, *params)`
Computes the Vandermonde matrix.

Parameters

x : ndarray

input

params : throw away parameter
parameter list returned by non-linear fitters

Returns

result : ndarray

The Vandermonde matrix

Chebyshev2D

```
class astropy.modeling.polynomial.Chebyshev2D(x_degree, y_degree, x_domain=None,
                                             x_window=[-1, 1], y_domain=None,
                                             y_window=[-1, 1], n_models=None,
                                             model_set_axis=None, **params)
```

Bases: `astropy.modeling.polynomial.OrthoPolynomialBase`

2D Chebyshev polynomial of the 1st kind.

It is defined as

$$P_{n_m}(x, y) = \sum C_{n_m} T_n(x) T_m(y)$$

Parameters

x_degree : int

degree in x

y_degree : int

degree in y

x_domain : list or None

domain of the x independent variable

y_domain : list or None

domain of the y independent variable

x_window : list or None

range of the x independent variable

y_window : list or None

range of the y independent variable

param_dim : int

number of parameter sets

****params** : dict

keyword: value pairs, representing parameter_name: value

Other Parameters

fixed : a dict

A dictionary {parameter_name: boolean} of parameters to not be varied during fitting. True means the parameter is held fixed. Alternatively the `fixed` property of a parameter may be used.

tied : dict

A dictionary {parameter_name: callable} of parameters which are linked to some other parameter. The dictionary values are callables providing the linking relationship. Alternatively the `tied` property of a parameter may be used.

bounds : dict

A dictionary {parameter_name: boolean} of lower and upper bounds of parameters. Keys are parameter names. Values are a list of length 2 giving the desired range for the parameter. Alternatively the `min` and `max` properties of a parameter may be used.

eqcons : list

A list of functions of length n such that `eqcons[j](x0, *args) == 0.0` in a successfully optimized problem.

ineqcons : list

A list of functions of length n such that `ieqcons[j](x0, *args) >= 0.0` is a successfully optimized problem.

Methods Summary

`fit_deriv(x, y, *params)` Derivatives with respect to the coefficients.

Methods Documentation

fit_deriv (*x*, *y*, **params*)

Derivatives with respect to the coefficients.

This is an array with Chebyshev polynomials:

$$T_{x_0} T_{y_0}, T_{x_1} T_{y_0} \dots T_{x_n} T_{y_0} \dots T_{x_n} T_{y_m}$$

Parameters

x : ndarray

input

y : ndarray

input

params : throw away parameter

parameter list returned by non-linear fitters

Returns

result : ndarray

The Vandermonde matrix

InverseSIP

class `astropy.modeling.polynomial.InverseSIP` (*ap_order*, *bp_order*, *ap_coeff*={}, *bp_coeff*={}, *n_models*=None, *model_set_axis*=None)

Bases: `astropy.modeling.Model`

Inverse Simple Imaging Polynomial

Parameters

ap_order : int

order for the inverse transformation (AP coefficients)

bp_order : int

order for the inverse transformation (BP coefficients)

ap_coeff : dict

coefficients for the inverse transform
bp_coeff : dict
coefficients for the inverse transform
param_dim : int
number of parameter sets

Attributes Summary

<code>n_inputs</code>	<code>int(x[, base]) -> integer</code>
<code>n_outputs</code>	<code>int(x[, base]) -> integer</code>

Methods Summary

<code>__call__(x, y)</code>

Attributes Documentation

n_inputs = 2

n_outputs = 2

Methods Documentation

`__call__(x, y)`

Legendre1D

```
class astropy.modeling.polynomial.Legendre1D(degree, domain=None, window=[-1, 1],
                                             n_models=None, model_set_axis=None,
                                             **params)
```

Bases: `astropy.modeling.polynomial.PolynomialModel`

1D Legendre polynomial.

Parameters

degree : int

degree of the series

domain : list or None

window : list or None

If None, it is set to [-1,1] Fitters will remap the domain to this window

param_dim : int

number of parameter sets

****params** : dict

keyword: value pairs, representing parameter_name: value

Other Parameters

fixed : a dict

A dictionary {parameter_name: boolean} of parameters to not be varied during fitting. True means the parameter is held fixed. Alternatively the `fixed` property of a parameter may be used.

tied : dict

A dictionary {parameter_name: callable} of parameters which are linked to some other parameter. The dictionary values are callables providing the linking relationship. Alternatively the `tied` property of a parameter may be used.

bounds : dict

A dictionary {parameter_name: boolean} of lower and upper bounds of parameters. Keys are parameter names. Values are a list of length 2 giving the desired range for the parameter. Alternatively the `min` and `max` properties of a parameter may be used.

eqcons : list

A list of functions of length `n` such that `eqcons[j](x0, *args) == 0.0` in a successfully optimized problem.

ineqcons : list

A list of functions of length `n` such that `ieqcons[j](x0, *args) >= 0.0` is a successfully optimized problem.

Methods Summary

<code>__call__(*inputs, **kwargs)</code>	Transforms data using this model.
<code>clenshaw(x, coeff)</code>	
<code>fit_deriv(x, *params)</code>	Computes the Vandermonde matrix.

Methods Documentation

`__call__(*inputs, **kwargs)`
Transforms data using this model.

Parameters

x : scalar, list or array

input

`clenshaw(x, coeff)`

`fit_deriv(x, *params)`
Computes the Vandermonde matrix.

Parameters

x : ndarray

input

params : throw away parameter
parameter list returned by non-linear fitters

Returns

result : ndarray
The Vandermonde matrix

Legendre2D

```
class astropy.modeling.polynomial.Legendre2D(x_degree, y_degree, x_domain=None,
                                             x_window=[-1, 1], y_domain=None,
                                             y_window=[-1, 1], n_models=None,
                                             model_set_axis=None, **params)
```

Bases: `astropy.modeling.polynomial.OrthoPolynomialBase`

Legendre 2D polynomial.

Defined as:

$$P_{nm}(x, y) = C_{nm} L_n(x) L_m(y)$$

Parameters

x_degree : int
degree in x

y_degree : int
degree in y

x_domain : list or None
domain of the x independent variable

y_domain : list or None
domain of the y independent variable

x_window : list or None
range of the x independent variable

y_window : list or None
range of the y independent variable

param_dim : int
number of parameter sets

****params** : dict
keyword: value pairs, representing parameter_name: value

Other Parameters

fixed : a dict
A dictionary {parameter_name: boolean} of parameters to not be varied during fitting. True means the parameter is held fixed. Alternatively the `fixed` property of a parameter may be used.

tied : dict

A dictionary `{parameter_name: callable}` of parameters which are linked to some other parameter. The dictionary values are callables providing the linking relationship. Alternatively the `tied` property of a parameter may be used.

bounds : dict

A dictionary `{parameter_name: boolean}` of lower and upper bounds of parameters. Keys are parameter names. Values are a list of length 2 giving the desired range for the parameter. Alternatively the `min` and `max` properties of a parameter may be used.

eqcons : list

A list of functions of length `n` such that `eqcons[j](x0, *args) == 0.0` in a successfully optimized problem.

ineqcons : list

A list of functions of length `n` such that `ineqcons[j](x0, *args) >= 0.0` is a successfully optimized problem.

Methods Summary

`fit_deriv(x, y, *params)` Derivatives with respect to the coefficients.

Methods Documentation

fit_deriv (*x*, *y*, **params*)

Derivatives with respect to the coefficients. This is an array with Legendre polynomials:

`Lx0Ly0 Lx1Ly0...LxnLy0...LxnLym`

Parameters

x : ndarray

input

y : ndarray

input

params : throw away parameter

parameter list returned by non-linear fitters

Returns

result : ndarray

The Vandermonde matrix

Polynomial1D

```
class astropy.modeling.polynomial.Polynomial1D (degree, domain=[-1, 1], window=[-1, 1],
                                                n_models=None, model_set_axis=None,
                                                **params)
```

Bases: `astropy.modeling.polynomial.PolynomialModel`

1D Polynomial model.

Parameters**degree** : int

degree of the series

domain : list or None**window** : list or None

If None, it is set to [-1,1] Fitters will remap the domain to this window

param_dim : int

number of parameter sets

****params** : dict

keyword: value pairs, representing parameter_name: value

Other Parameters**fixed** : a dict

A dictionary {parameter_name: boolean} of parameters to not be varied during fitting. True means the parameter is held fixed. Alternatively the `fixed` property of a parameter may be used.

tied : dict

A dictionary {parameter_name: callable} of parameters which are linked to some other parameter. The dictionary values are callables providing the linking relationship. Alternatively the `tied` property of a parameter may be used.

bounds : dict

A dictionary {parameter_name: boolean} of lower and upper bounds of parameters. Keys are parameter names. Values are a list of length 2 giving the desired range for the parameter. Alternatively the `min` and `max` properties of a parameter may be used.

eqcons : list

A list of functions of length n such that `eqcons[j](x0, *args) == 0.0` in a successfully optimized problem.

ineqcons : list

A list of functions of length n such that `ineqcons[j](x0, *args) >= 0.0` is a successfully optimized problem.

Methods Summary

<code>__call__(*inputs, **kwargs)</code>	Transforms data using this model.
<code>fit_deriv(x, *params)</code>	Computes the Vandermonde matrix.
<code>horner(x, coef)</code>	

Methods Documentation`__call__(*inputs, **kwargs)`

Transforms data using this model.

Parameters

x : scalar, list or array

input

fit_deriv (*x*, **params*)

Computes the Vandermonde matrix.

Parameters

x : ndarray

input

params : throw away parameter

parameter list returned by non-linear fitters

Returns

result : ndarray

The Vandermonde matrix

horner (*x*, *coef*)

Polynomial2D

class `astropy.modeling.polynomial.Polynomial2D` (*degree*, *x_domain*=[-1, 1], *y_domain*=[-1, 1], *x_window*=[-1, 1], *y_window*=[-1, 1], *n_models*=None, *model_set_axis*=None, ***params*)

Bases: `astropy.modeling.polynomial.PolynomialModel`

2D Polynomial model.

Represents a general polynomial of degree n:

$$P(x, y) = c_{00} + c_{10}x + \dots + c_{n0}x^n + c_{01}y + \dots + c_{0n}y^n + c_{11}xy + c_{12}xy^2 + \dots + c_{1(n-1)}xy^{n-1} + \dots + c_{(n-1)1}x^{n-1}y$$

Parameters

degree : int

highest power of the polynomial, the number of terms is degree+1

x_domain : list or None

domain of the x independent variable

y_domain : list or None

domain of the y independent variable

x_window : list or None

range of the x independent variable

y_window : list or None

range of the y independent variable

param_dim : int

number of parameter sets

****params** : dict

keyword: value pairs, representing parameter_name: value

Other Parameters**fixed** : dict

A dictionary {`parameter_name`: `boolean`} of parameters to not be varied during fitting. True means the parameter is held fixed. Alternatively the `fixed` property of a parameter may be used.

tied : dict

A dictionary {`parameter_name`: `callable`} of parameters which are linked to some other parameter. The dictionary values are callables providing the linking relationship. Alternatively the `tied` property of a parameter may be used.

bounds : dict

A dictionary {`parameter_name`: `boolean`} of lower and upper bounds of parameters. Keys are parameter names. Values are a list of length 2 giving the desired range for the parameter. Alternatively the `min` and `max` properties of a parameter may be used.

eqcons : list

A list of functions of length `n` such that `eqcons[j](x0, *args) == 0.0` in a successfully optimized problem.

ineqcons : list

A list of functions of length `n` such that `ieqcons[j](x0, *args) >= 0.0` is a successfully optimized problem.

Methods Summary

<code>__call__(*inputs, **kwargs)</code>	Transforms data using this model.
<code>fit_deriv(x, y, *params)</code>	Computes the Vandermonde matrix.
<code>invlex_coeff()</code>	
<code>mhorners(x, y, coeff)</code>	Multivariate Horner's scheme

Methods Documentation`__call__(*inputs, **kwargs)`

Transforms data using this model.

Parameters`x` : scalar, list or array

input

`y` : scalar, list or array

input

`fit_deriv(x, y, *params)`

Computes the Vandermonde matrix.

Parameters`x` : ndarray

input

`y` : ndarray

input

params : throw away parameter

parameter list returned by non-linear fitters

Returns

result : ndarray

The Vandermonde matrix

`invlex_coeff()`

`mhorner(x, y, coeff)`

Multivariate Horner's scheme

Parameters

x, y : array

coeff : array of coefficients in inverse lexical order

SIP

`class astropy.modeling.polynomial.SIP` (*crpix*, *a_order*, *b_order*, *a_coeff*={}, *b_coeff*={},
ap_order=None, *bp_order*=None, *ap_coeff*={},
bp_coeff={}, *n_models*=None, *model_set_axis*=None)

Bases: `astropy.modeling.Model`

Simple Imaging Polynomial (SIP) model.

The SIP convention is used to represent distortions in FITS image headers. See [R9] for a description of the SIP convention.

Parameters

crpix : list or ndarray of length(2)

CRPIX values

a_order : int

SIP polynomial order for first axis

b_order : int

SIP order for second axis

a_coeff : dict

SIP coefficients for first axis

b_coeff : dict

SIP coefficients for the second axis

ap_order : int

order for the inverse transformation (AP coefficients)

bp_order : int

order for the inverse transformation (BP coefficients)

ap_coeff : dict

coefficients for the inverse transform

bp_coeff : dict
 coefficients for the inverse transform

param_dim : int
 number of parameter sets

References

[R9]

Attributes Summary

<code>n_inputs</code>	<code>int(x[, base]) -> integer</code>
<code>n_outputs</code>	<code>int(x[, base]) -> integer</code>

Methods Summary

<code>__call__(x, y)</code>
<code>inverse()</code>

Attributes Documentation

n_inputs = 2

n_outputs = 2

Methods Documentation

__call__(x, y)

inverse()

OrthoPolynomialBase

```
class astropy.modeling.polynomial.OrthoPolynomialBase(x_degree, y_degree,
                                                       x_domain=None,
                                                       x_window=None,
                                                       y_domain=None,
                                                       y_window=None,
                                                       n_models=None,
                                                       model_set_axis=None,
                                                       **params)
```

Bases: `astropy.modeling.polynomial.PolynomialBase`

This is a base class for the 2D Chebyshev and Legendre models.

The polynomials implemented here require a maximum degree in x and y.

Parameters

x_degree : int

degree in x

y_degree : int

degree in y

x_domain : list or None

domain of the x independent variable

x_window : list or None

range of the x independent variable

y_domain : list or None

domain of the y independent variable

y_window : list or None

range of the y independent variable

param_dim : int

number of parameter sets

****params** : dict

{keyword: value} pairs, representing {parameter_name: value}

Attributes Summary

<code>n_inputs</code>	<code>int(x[, base]) -> integer</code>
<code>n_outputs</code>	<code>int(x[, base]) -> integer</code>

Methods Summary

<code>__call__(*inputs, **kwargs)</code>	Transforms data using this model.
<code>get_num_coeff()</code>	Determine how many coefficients are needed
<code>imhorner(x, y, coeff)</code>	
<code>invlex_coeff()</code>	

Attributes Documentation

`n_inputs = 2`

`n_outputs = 1`

Methods Documentation

`__call__` (*inputs, **kwargs)

Transforms data using this model.

Parameters

x : scalar, list or array

y : scalar, list or array

`get_num_coeff` ()

Determine how many coefficients are needed

Returns

numc : int

number of coefficients

`imhorner` (x, y, coeff)

`invlex_coeff` ()

PolynomialModel

```
class astropy.modeling.polynomial.PolynomialModel (degree, n_inputs=1,
                                                    n_outputs=1, n_models=None,
                                                    model_set_axis=None, **params)
```

Bases: `astropy.modeling.polynomial.PolynomialBase`

Base class for polynomial models.

Its main purpose is to determine how many coefficients are needed based on the polynomial order and dimension and to provide their default values, names and ordering.

Attributes Summary

<code>degree</code>	Degree of polynomial.
<code>n_inputs</code>	The number of input variables to model evaluation.
<code>n_outputs</code>	The number of outputs returned when model is evaluated.

Methods Summary

<code>get_num_coeff(ndim)</code>	Return the number of coefficients in one parameter set
----------------------------------	--

Attributes Documentation

`degree`

Degree of polynomial.

`n_inputs`

The number of input variables to model evaluation.

n_outputs

The number of outputs returned when model is evaluated.

Methods Documentation**get_num_coeff** (*ndim*)

Return the number of coefficients in one parameter set

Class Inheritance Diagram**12.4.7 astropy.modeling.projections Module**

Implements projections—particularly sky projections defined in WCS Paper II [R19]

All angles are set and displayed in degrees but internally computations are performed in radians.

References**Classes**

<code>Pix2Sky_AZP(mu, gamma)</code>	AZP : Zenital perspective projection - pixel to sky.
<code>Sky2Pix_AZP(mu, gamma)</code>	AZP : Zenital perspective projection - sky to pixel.
<code>Pix2Sky_CAR(*args, **kwargs)</code>	CAR: Plate carree projection - pixel to sky.
<code>Sky2Pix_CAR(*args, **kwargs)</code>	CAR: Plate carree projection - sky to pixel.
<code>Pix2Sky_CEA(lam)</code>	CEA : Cylindrical equal area projection - pixel to sky.
<code>Sky2Pix_CEA(lam)</code>	CEA: Cylindrical equal area projection - sky to pixel.
<code>Pix2Sky_CYP(mu, lam)</code>	CYP : Cylindrical perspective - pixel to sky.
<code>Sky2Pix_CYP(mu, lam)</code>	CYP : Cylindrical Perspective - sky to pixel.
<code>Pix2Sky_MER(*args, **kwargs)</code>	MER: Mercator - pixel to sky.
<code>Sky2Pix_MER(*args, **kwargs)</code>	MER: Mercator - sky to pixel.
<code>Pix2Sky_SIN(*args, **kwargs)</code>	SIN : Slant orthographic projection - pixel to sky.
<code>Sky2Pix_SIN(*args, **kwargs)</code>	SIN : Slant orthographic projection - sky to pixel.
<code>Pix2Sky_STG(*args, **kwargs)</code>	STG : Stereographic Projection - pixel to sky.
<code>Sky2Pix_STG(*args, **kwargs)</code>	STG : Stereographic Projection - sky to pixel.
<code>Pix2Sky_TAN(*args, **kwargs)</code>	TAN : Gnomonic projection - pixel to sky.
<code>Sky2Pix_TAN(*args, **kwargs)</code>	TAN : Gnomonic Projection - sky to pixel.
<code>AffineTransformation2D(matrix, translation)</code>	Perform an affine transformation in 2 dimensions.

Pix2Sky_AZP

class `astropy.modeling.projections.Pix2Sky_AZP` (*mu=0.0, gamma=0.0*)

Bases: `astropy.modeling.projections.Zenithal`

AZP : Zenital perspective projection - pixel to sky.

Parameters

mu : float

distance from point of projection to center of sphere in spherical radii, default is 0.

gamma : float

look angle in deg, default is 0.

Attributes Summary

```
gamma
mu
param_names  list() -> new empty list
```

Methods Summary

```
__call__(*inputs, **kwargs)
check_mu(val)
inverse()
```

Attributes Documentation

gamma

mu

param_names = ['mu', 'gamma']

Methods Documentation

__call__ (*inputs, **kwargs)

check_mu (val)

inverse ()

Sky2Pix_AZP

class astropy.modeling.projections.**Sky2Pix_AZP** (*mu=0.0, gamma=0.0*)

Bases: astropy.modeling.projections.Zenithal

AZP : Zenital perspective projection - sky to pixel.

Parameters

mu : float

distance from point of projection to center of sphere in spherical radii, default is 0.

gamma : float

look angle in deg, default is 0.

Attributes Summary

```
gamma
mu
param_names list() -> new empty list
```

Methods Summary

```
__call__(*inputs, **kwargs)
check_mu(val)
inverse()
```

Attributes Documentation

gamma

mu

param_names = ['mu', 'gamma']

Methods Documentation

__call__ (*inputs, **kwargs)

check_mu (val)

inverse ()

Pix2Sky_CAR

class astropy.modeling.projections.**Pix2Sky_CAR** (*args, **kwargs)

Bases: astropy.modeling.projections.Cylindrical

CAR: Plate carree projection - pixel to sky.

Methods Summary

```
__call__(*inputs, **kwargs)
inverse()
```

Methods Documentation

__call__ (*inputs, **kwargs)

```
inverse()
```

Sky2Pix_CAR

```
class astropy.modeling.projections.Sky2Pix_CAR(*args, **kwargs)
    Bases: astropy.modeling.projections.Cylindrical
    CAR: Plate carree projection - sky to pixel.
```

Methods Summary

```
__call__(*inputs, **kwargs)
inverse()
```

Methods Documentation

```
__call__(*inputs, **kwargs)
```

```
inverse()
```

Pix2Sky_CEA

```
class astropy.modeling.projections.Pix2Sky_CEA(lam=1)
    Bases: astropy.modeling.projections.Cylindrical
    CEA : Cylindrical equal area projection - pixel to sky.
```

Attributes Summary

```
lam
param_names list() -> new empty list
```

Methods Summary

```
__call__(x, y)
inverse()
```

Attributes Documentation

```
lam
```

```
param_names = ['lam']
```

Methods Documentation

`__call__(x, y)`

`inverse()`

Sky2Pix_CEA

`class astropy.modeling.projections.Sky2Pix_CEA(lam=1)`
Bases: `astropy.modeling.projections.Cylindrical`
CEA: Cylindrical equal area projection - sky to pixel.

Attributes Summary

<code>lam</code>
<code>param_names</code> list() -> new empty list

Methods Summary

<code>__call__(*inputs, **kwargs)</code>
<code>inverse()</code>

Attributes Documentation

`lam`

`param_names = ['lam']`

Methods Documentation

`__call__(*inputs, **kwargs)`

`inverse()`

Pix2Sky_CYP

`class astropy.modeling.projections.Pix2Sky_CYP(mu, lam)`
Bases: `astropy.modeling.projections.Cylindrical`
CYP : Cylindrical perspective - pixel to sky.

Attributes Summary

```
lam
mu
param_names  list() -> new empty list
```

Methods Summary

```
__call__(*inputs, **kwargs)
inverse()
```

Attributes Documentation**lam****mu****param_names = ['mu', 'lam']****Methods Documentation****__call__** (*inputs, **kwargs)**inverse** ()**Sky2Pix_CYP****class** astropy.modeling.projections.**Sky2Pix_CYP** (*mu, lam*)

Bases: astropy.modeling.projections.Cylindrical

CYP : Cylindrical Perspective - sky to pixel.

Attributes Summary

```
lam
mu
param_names  list() -> new empty list
```

Methods Summary

```
__call__(*inputs, **kwargs)
inverse()
```

Attributes Documentation

`lam`

`mu`

`param_names = ['mu', 'lam']`

Methods Documentation

`__call__(*inputs, **kwargs)`

`inverse()`

Pix2Sky_MER

`class astropy.modeling.projections.Pix2Sky_MER(*args, **kwargs)`

Bases: `astropy.modeling.projections.Cylindrical`

MER: Mercator - pixel to sky.

Methods Summary

`__call__(*inputs, **kwargs)`
`inverse()`

Methods Documentation

`__call__(*inputs, **kwargs)`

`inverse()`

Sky2Pix_MER

`class astropy.modeling.projections.Sky2Pix_MER(*args, **kwargs)`

Bases: `astropy.modeling.projections.Cylindrical`

MER: Mercator - sky to pixel.

Methods Summary

`__call__(*inputs, **kwargs)`

Continued on next page

Table 12.117 – continued from previous page

inverse()**Methods Documentation**`__call__ (*inputs, **kwargs)``inverse ()`**Pix2Sky_SIN**`class astropy.modeling.projections.Pix2Sky_SIN (*args, **kwargs)`Bases: `astropy.modeling.projections.Zenithal`

SIN : Slant orthographic projection - pixel to sky.

Methods Summary`__call__ (*inputs, **kwargs)``inverse()`**Methods Documentation**`__call__ (*inputs, **kwargs)``inverse ()`**Sky2Pix_SIN**`class astropy.modeling.projections.Sky2Pix_SIN (*args, **kwargs)`Bases: `astropy.modeling.projections.Zenithal`

SIN : Slant orthographic projection - sky to pixel.

Methods Summary`__call__(phi, theta)``inverse()`**Methods Documentation**`__call__ (phi, theta)``inverse ()`

Pix2Sky_STG

class `astropy.modeling.projections.Pix2Sky_STG(*args, **kwargs)`
Bases: `astropy.modeling.projections.Zenithal`
STG : Stereographic Projection - pixel to sky.

Methods Summary

`__call__(x, y)`
`inverse()`

Methods Documentation

`__call__(x, y)`

`inverse()`

Sky2Pix_STG

class `astropy.modeling.projections.Sky2Pix_STG(*args, **kwargs)`
Bases: `astropy.modeling.projections.Zenithal`
STG : Stereographic Projection - sky to pixel.

Methods Summary

`__call__(*inputs, **kwargs)`
`inverse()`

Methods Documentation

`__call__(*inputs, **kwargs)`

`inverse()`

Pix2Sky_TAN

class `astropy.modeling.projections.Pix2Sky_TAN(*args, **kwargs)`
Bases: `astropy.modeling.projections.Zenithal`
TAN : Gnomonic projection - pixel to sky.

Methods Summary

```
__call__(*inputs, **kwargs)
inverse()
```

Methods Documentation

```
__call__ (*inputs, **kwargs)
```

```
inverse ()
```

Sky2Pix_TAN

```
class astropy.modeling.projections.Sky2Pix_TAN (*args, **kwargs)
```

Bases: `astropy.modeling.projections.Zenithal`

TAN : Gnomonic Projection - sky to pixel.

Methods Summary

```
__call__(*inputs, **kwargs)
inverse()
```

Methods Documentation

```
__call__ (*inputs, **kwargs)
```

```
inverse ()
```

AffineTransformation2D

```
class astropy.modeling.projections.AffineTransformation2D (matrix=[[1.0, 0.0], [0.0,
                                                                1.0]], translation=[0.0,
                                                                0.0])
```

Bases: `astropy.modeling.Model`

Perform an affine transformation in 2 dimensions.

Parameters

matrix : array

A 2x2 matrix specifying the linear transformation to apply to the inputs

translation : array

A 2D vector (given as either a 2x1 or 1x2 array) specifying a translation to apply to the inputs

Attributes Summary

<code>matrix</code>	
<code>n_inputs</code>	<code>int(x[, base]) -> integer</code>
<code>n_outputs</code>	<code>int(x[, base]) -> integer</code>
<code>param_names</code>	<code>list() -> new empty list</code>
<code>standard_broadcasting</code>	<code>bool(x) -> bool</code>
<code>translation</code>	

Methods Summary

<code>__call__(*inputs, **kwargs)</code>	Apply the transformation to a set of 2D Cartesian coordinates given as two lists—one for the x coordinates and one for the y coordinates—or a single coordinate pair.
<code>inverse()</code>	Inverse transformation.

Attributes Documentation

matrix

n_inputs = 2

n_outputs = 2

param_names = ['matrix', 'translation']

standard_broadcasting = False

translation

Methods Documentation

__call__ (*inputs, **kwargs)

Apply the transformation to a set of 2D Cartesian coordinates given as two lists—one for the x coordinates and one for the y coordinates—or a single coordinate pair.

Parameters

x, y : array, float

x and y coordinates

inverse ()

Inverse transformation.

Raises `InputParameterError` if the transformation cannot be inverted.

Class Inheritance Diagram

12.4.8 astropy.modeling.statistic Module

Statistic functions used in *fitting*.

Functions

`leastsquare(measured_vals, updated_model, ...)` Least square statistic with optional weights.

leastsquare

`astropy.modeling.statistic.leastsquare` (*measured_vals*, *updated_model*, *weights*, *x*,
y=None)

Least square statistic with optional weights.

Parameters

measured_vals : `ndarray`

Measured data values.

updated_model : `Model`

Model with parameters set by the current iteration of the optimizer.

weights : `ndarray`

Array of weights to apply to each residual.

x : `ndarray`

Independent variable “x” to evaluate the model on.

y : `ndarray`, optional

Independent variable “y” to evaluate the model on, for 2D models.

Returns

res : `float`

The sum of least squares.

12.4.9 astropy.modeling.rotations Module

Implements rotations, including spherical rotations as defined in WCS Paper II [R20]

`RotateNative2Celestial` and `RotateCelestial2Native` follow the convention in WCS Paper II to rotate to/from a native sphere and the celestial sphere.

The user interface sets and displays angles in degrees but the values are stored internally in radians. This is managed through the parameter setters/getters.

References

Classes

<code>RotateCelestial2Native(phi, theta, psi)</code>	Transformation from Celestial to Native to Spherical Coordinates.
<code>RotateNative2Celestial(phi, theta, psi)</code>	Transformation from Native to Celestial Spherical Coordinates.
<code>Rotation2D([angle])</code>	Perform a 2D rotation given an angle in degrees.

RotateCelestial2Native

class `astropy.modeling.rotations.RotateCelestial2Native` (*phi, theta, psi*)

Bases: `astropy.modeling.rotations.EulerAngleRotation`

Transformation from Celestial to Native to Spherical Coordinates.

Defines a ZXZ rotation.

Parameters

phi, theta, psi : float

Euler angles in deg

Methods Summary

`__call__(alpha, cdelta)`
`inverse()`

Methods Documentation

`__call__` (*alpha, cdelta*)

`inverse()`

RotateNative2Celestial

class `astropy.modeling.rotations.RotateNative2Celestial` (*phi, theta, psi*)

Bases: `astropy.modeling.rotations.EulerAngleRotation`

Transformation from Native to Celestial Spherical Coordinates.

Defines a ZXZ rotation.

Parameters

phi, theta, psi : float

Euler angles in deg

Methods Summary

`__call__(nphi, ntheta)`
`inverse()`

Methods Documentation

`__call__` (*nphi*, *ntheta*)

`inverse` ()

Rotation2D

class `astropy.modeling.rotations.Rotation2D` (*angle=0.0*)

Bases: `astropy.modeling.Model`

Perform a 2D rotation given an angle in degrees.

Positive angles represent a counter-clockwise rotation and vice-versa.

Parameters

angle : float

angle of rotation in deg

Attributes Summary

<code>angle</code>	
<code>n_inputs</code>	<code>int(x[, base]) -> integer</code>
<code>n_outputs</code>	<code>int(x[, base]) -> integer</code>
<code>param_names</code>	<code>list() -> new empty list</code>

Methods Summary

<code>__call__(x, y)</code>	Apply the rotation to a set of 2D Cartesian coordinates given as two lists—one for the x coordinates and one for a y
<code>inverse()</code>	Inverse rotation.

Attributes Documentation

angle

`n_inputs = 2`

`n_outputs = 2`

`param_names = ['angle']`

Methods Documentation

`__call__` (*x*, *y*)

Apply the rotation to a set of 2D Cartesian coordinates given as two lists—one for the x coordinates and

one for a y coordinates—or a single coordinate pair.

Parameters

x, y : array, float

x and y coordinates

inverse ()

Inverse rotation.

Class Inheritance Diagram

Connecting up: Files and I/O

UNIFIED FILE READ/WRITE INTERFACE

Astropy provides a unified interface for reading and writing data in different formats. For many common cases this will simplify the process of file I/O and reduce the need to master the separate details of all the I/O packages within Astropy. This functionality is still in active development and the number of supported formats will be increasing. For details on the implementation see *I/O Registry* ([astropy.io.registry](#)).

13.1 Getting started with Table I/O

The `Table` class includes two methods, `read()` and `write()`, that make it possible to read from and write to files. A number of formats are automatically supported (see [Built-in table readers/writers](#)) and new file formats and extensions can be registered with the `Table` class (see *I/O Registry* ([astropy.io.registry](#))).

To use this interface, first import the `Table` class, then simply call the `Table.read()` method with the name of the file and the file format, for instance `'ascii.daophot'`:

```
>>> from astropy.table import Table
>>> t = Table.read('photometry.dat', format='ascii.daophot')
```

It is possible to load tables directly from the Internet using URLs. For example, download tables from VizieR catalogues in CDS format (`'ascii.cds'`):

```
>>> t = Table.read("ftp://cdsarc.u-strasbg.fr/pub/cats/VII/253/snrs.dat",
...               readme="ftp://cdsarc.u-strasbg.fr/pub/cats/VII/253/ReadMe",
...               format="ascii.cds")
```

For certain file formats, the format can be automatically detected, for example from the filename extension:

```
>>> t = Table.read('table.tex')
```

Similarly, for writing, the format can be explicitly specified:

```
>>> t.write(filename, format='latex')
```

As for the `read()` method, the format may be automatically identified in some cases.

Any additional arguments specified will depend on the format. For examples of this see the section [Built-in table readers/writers](#). This section also provides the full list of choices for the `format` argument.

13.2 Built-in table readers/writers

The full list of built-in readers and writers is shown in the table below:

Format	Read	Write	Auto-identify	Deprecated
aastex	Yes	Yes	No	Yes
ascii	Yes	Yes	No	
ascii.aastex	Yes	Yes	No	
ascii.basic	Yes	Yes	No	
ascii.cds	Yes	No	No	
ascii.commented_header	Yes	Yes	No	
ascii.daophot	Yes	No	No	
ascii.fixed_width	Yes	Yes	No	
ascii.fixed_width_no_header	Yes	Yes	No	
ascii.fixed_width_two_line	Yes	Yes	No	
ascii.html	Yes	Yes	Yes	
ascii.ipac	Yes	Yes	No	
ascii.latex	Yes	Yes	Yes	
ascii.no_header	Yes	Yes	No	
ascii.rdb	Yes	Yes	Yes	
ascii.sextractor	Yes	No	No	
ascii.tab	Yes	Yes	No	
ascii.csv	Yes	Yes	Yes	
cds	Yes	No	No	Yes
daophot	Yes	No	No	Yes
fits	Yes	Yes	Yes	
hdf5	Yes	Yes	Yes	
html	Yes	Yes	No	Yes
ipac	Yes	Yes	No	Yes
latex	Yes	Yes	No	Yes
rdb	Yes	Yes	No	Yes
votable	Yes	Yes	Yes	

Deprecated format names like `aastex` will be removed in a future version. Use the full name (e.g. `ascii.aastex`) instead.

13.2.1 ASCII formats

The `read()` and `write()` methods can be used to read and write formats supported by `astropy.io.ascii`.

Use `format='ascii'` in order to interface to the generic `read()` and `write()` functions from `astropy.io.ascii`. When reading a table this means that all supported ASCII table formats will be tried in order to successfully parse the input. For example:

```
>>> t = Table.read('astropy/io/ascii/tests/t/latex1.tex', format='ascii')
>>> print t
cola colb colc
----
a      1      2
b      3      4
```

When writing a table with `format='ascii'` the output is a basic character-delimited file with a single header line containing the column names.

All additional arguments are passed to the `astropy.io.ascii.read()` and `write()` functions. Further details are available in the sections on [Parameters for read\(\)](#) and [Parameters for write\(\)](#). For example, to change column delimiter and the output format for the `colc` column use:

```
>>> t.write(sys.stdout, format='ascii', delimiter='|', formats={'colc': '%0.2f'})
cola|colb|colc
```

```
a|1|2.00
b|3|4.00
```

A full list of the supported `format` values and corresponding format types for ASCII tables is given below. The `Suffix` column indicates the filename suffix where the format will be auto-detected, while the `Write` column indicates which support write functionality.

Format	Suffix	Write	Description
<code>ascii</code>		Yes	ASCII table in any supported format (uses guessing)
<code>ascii.aastex</code>		Yes	<code>AASSTex</code> : AASSTeX deluxetable used for AAS journals
<code>ascii.basic</code>		Yes	<code>Basic</code> : Basic table with custom delimiters
<code>ascii.cds</code>			<code>Cds</code> : CDS format table
<code>ascii.comments_header</code>		Yes	<code>CommentsHeader</code> : Column names in a commented line
<code>ascii.daophot</code>			<code>Daophot</code> : IRAF DAOPHOT format table
<code>ascii.fixed_width</code>		Yes	<code>FixedWidth</code> : Fixed width
<code>ascii.fixed_width_no_header</code>		Yes	<code>FixedWidthNoHeader</code> : Fixed width with no header
<code>ascii.fixed_width_two_line</code>		Yes	<code>FixedWidthTwoLine</code> : Fixed width with second header line
<code>ascii.ipac</code>		Yes	<code>Ipac</code> : IPAC format table
<code>ascii.html</code>	<code>.html</code>	Yes	<code>HTML</code> : HTML table
<code>ascii.latex</code>	<code>.tex</code>	Yes	<code>Latex</code> : LaTeX table
<code>ascii.no_header</code>		Yes	<code>NoHeader</code> : Basic table with no headers
<code>ascii.rdb</code>	<code>.rdb</code>	Yes	<code>Rdb</code> : Tab-separated with a type definition header line
<code>ascii.sextractor</code>			<code>SEXTRACTOR</code> : SExtractor format table
<code>ascii.tab</code>		Yes	<code>Tab</code> : Basic table with tab-separated values
<code>ascii.csv</code>	<code>.csv</code>	Yes	<code>Csv</code> : Basic table with comma-separated values

Note: When specifying a specific ASCII table format using the unified interface, the format name is prefixed with `ascii.` in order to identify the format as ASCII-based. Compare the table above to the `astropy.io.ascii` list of *Supported formats*. Therefore the following are equivalent:

```
>>> dat = ascii.read('file.dat', format='daophot')
>>> dat = Table.read('file.dat', format='ascii.daophot')
```

For compatibility with `astropy` version 0.2 and earlier, the following format values are also allowed in `Table.read()`: `daophot`, `ipac`, `html`, `latex`, and `rdb`.

13.2.2 FITS

Reading/writing from/to `FITS` files is supported with `format='fits'`. In most cases, existing `FITS` files should be automatically identified as such based on the header of the file, but if not, or if writing to disk, then the format should be explicitly specified.

If a `FITS` table file contains only a single table, then it can be read in with:

```
>>> t = Table.read('data.fits')
```

If more than one table is present in the file, the first table found will be read in and a warning will be emitted:

```
>>> t = Table.read('data.fits')
WARNING: hdu= was not specified but multiple tables are present, reading in first available table (h
```

To write to a new file:

```
>>> t.write('new_table.fits')
```

At this time, the `meta` attribute of the `Table` class is simply an ordered dictionary and does not fully represent the structure of a FITS header (for example, keyword comments are dropped). This is likely to change in a future release.

13.2.3 HDF5

Reading/writing from/to **HDF5** files is supported with `format='hdf5'` (this requires `h5py` to be installed). However, the `.hdf5` file extension is automatically recognized when writing files, and HDF5 files are automatically identified (even with a different extension) when reading in (using the first few bytes of the file to identify the format), so in most cases you will not need to explicitly specify `format='hdf5'`.

Since HDF5 files can contain multiple tables, the full path to the table should be specified via the `path=` argument when reading and writing. For example, to read a table called `data` from an HDF5 file named `observations.hdf5`, you can do:

```
>>> t = Table.read('observations.hdf5', path='data')
```

To read a table nested in a group in the HDF5 file, you can do:

```
>>> t = Table.read('observations.hdf5', path='group/data')
```

To write a table to a new file, the path should also be specified:

```
>>> t.write('new_file.hdf5', path='updated_data')
```

It is also possible to write a table to an existing file using `append=True`:

```
>>> t.write('observations.hdf5', path='updated_data', append=True)
```

As with other formats, the `overwrite=True` argument is supported for overwriting existing files. To overwrite only a single table within an HDF5 file that has multiple datasets, use *both* the `overwrite=True` and `append=True` arguments.

Finally, when writing to HDF5 files, the `compression=` argument can be used to ensure that the data is compressed on disk:

```
>>> t.write('new_file.hdf5', path='updated_data', compression=True)
```

13.2.4 VO Tables

Reading/writing from/to **VO table** files is supported with `format='votable'`. In most cases, existing VO tables should be automatically identified as such based on the header of the file, but if not, or if writing to disk, then the format should be explicitly specified.

If a VO table file contains only a single table, then it can be read in with:

```
>>> t = Table.read('aj285677t3_votable.xml')
```

If more than one table is present in the file, an error will be raised, unless the table ID is specified via the `table_id=` argument:

```
>>> t = Table.read('catalog.xml')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Volumes/Raptor/Library/Python/2.7/lib/python/site-packages/astropy/table/table.py", line 153
    table = reader(*args, **kwargs)
```

```
File "/Volumes/Raptor/Library/Python/2.7/lib/python/site-packages/astropy/io/votable/connect.py", line 155
    raise ValueError("Multiple tables found: table id should be set via the id= argument. The available tables are: %s" % available_tables)
ValueError: Multiple tables found: table id should be set via the table_id= argument. The available tables are: %s

>>> t = Table.read('catalog.xml', table_id='twomass')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Volumes/Raptor/Library/Python/2.7/lib/python/site-packages/astropy/table/table.py", line 155, in read
    table = reader(*args, **kwargs)
  File "/Volumes/Raptor/Library/Python/2.7/lib/python/site-packages/astropy/io/votable/connect.py", line 155, in read
    raise ValueError("Multiple tables found: table id should be set via the id= argument. The available tables are: %s" % available_tables)
ValueError: Multiple tables found: table id should be set via the table_id= argument. The available tables are: %s
```

To write to a new file, the ID of the table should also be specified (unless `t.meta['ID']` is defined):

```
>>> t.write('new_catalog.xml', table_id='updated_table', format='votable')
```

When writing, the `compression=True` argument can be used to force compression of the data on disk, and the `overwrite=True` argument can be used to overwrite an existing file.

FITS FILE HANDLING (ASTROPY.IO.FITS)

14.1 Introduction

The `astropy.io.fits` package provides access to FITS files. FITS (Flexible Image Transport System) is a portable file standard widely used in the astronomy community to store images and tables.

14.2 Getting Started

This section provides a quick introduction of using `astropy.io.fits`. The goal is to demonstrate the package's basic features without getting into too much detail. If you are a first time user or have never used Astropy or PyFITS, this is where you should start. See also the [FAQ](#) for answers to common questions/issues.

14.2.1 Reading and Updating Existing FITS Files

Opening a FITS file

Once the `astropy.io.fits` package is loaded using the standard convention¹, we can open an existing FITS file:

```
>>> from astropy.io import fits
>>> hdulist = fits.open('input.fits')
```

The `open()` function has several optional arguments which will be discussed in a later chapter. The default mode, as in the above example, is “readonly”. The `open` function returns an object called an `HDUList` which is a `list`-like collection of HDU objects. An HDU (Header Data Unit) is the highest level component of the FITS file structure, consisting of a header and (typically) a data array or table.

After the above `open` call, `hdulist[0]` is the primary HDU, `hdulist[1]` is the first extension HDU, etc (if there are any extensions), and so on. It should be noted that Astropy is using zero-based indexing when referring to HDUs and header cards, though the FITS standard (which was designed with FORTRAN in mind) uses one-based indexing.

The `HDUList` has a useful method `HDUList.info()`, which summarizes the content of the opened FITS file:

```
>>> hdulist.info()
Filename: test1.fits
No. Name Type Cards Dimensions Format
0 PRIMARY PrimaryHDU 220 () int16
1 SCI ImageHDU 61 (800, 800) float32
2 SCI ImageHDU 61 (800, 800) float32
3 SCI ImageHDU 61 (800, 800) float32
4 SCI ImageHDU 61 (800, 800) float32
```

¹ For legacy code only that already depends on PyFITS, it's acceptable to continue using “from astropy.io import fits as pyfits”.

After you are done with the opened file, close it with the `HDUList.close()` method:

```
>>> hdulist.close()
```

The headers will still be accessible after the `HDUList` is closed. The data may or may not be accessible depending on whether the data are touched and if they are memory-mapped, see later chapters for detail.

Working with large files

The `open()` function supports a `memmap=True` argument that allows the array data of each HDU to be accessed with `mmap`, rather than being read into memory all at once. This is particularly useful for working with very large arrays that cannot fit entirely into physical memory.

This has minimal impact on smaller files as well, though some operations, such as reading the array data sequentially, may incur some additional overhead. On 32-bit systems arrays larger than 2-3 GB cannot be `mmap`'d (which is fine, because by that point you're likely to run out of physical memory anyways), but 64-bit systems are much less limited in this respect.

Warning: When opening a file with `memmap=True`, because of how `mmap` works this means that when the HDU data is accessed (i.e. `hdul[0].data`) another handle to the FITS file is opened by `mmap`. This means that even after calling `hdul.close()` the `mmap` still holds an open handle to the data so that it can still be accessed by unwary programs that were built with the assumption that the `.data` attribute has all the data in-memory. In order to force the `mmap` to close either wait for the containing `HDUList` object to go out of scope, or manually call `del hdul[0].data` (this works so long as there are no other references held to the data array).

Working with FITS Headers

As mentioned earlier, each element of an `HDUList` is an HDU object with `.header` and `.data` attributes, which can be used to access the header and data portions of the HDU.

For those unfamiliar with FITS headers, they consist of a list of 80 byte “cards”, where a card contains a keyword, a value, and a comment. The keyword and comment must both be strings, whereas the value can be a string or an integer, floating point number, complex number, or `True/False`. Keywords are usually unique within a header, except in a few special cases.

The header attribute is a `Header` instance, another Astropy object. To get the value associated with a header keyword, simply do (a la Python dicts):

```
>>> hdulist[0].header['targname']
'NGC121'
```

to get the value of the keyword `targname`, which is a string ‘NGC121’.

Although keyword names are always in upper case inside the FITS file, specifying a keyword name with Astropy is case-insensitive, for the user’s convenience. If the specified keyword name does not exist, it will raise a `KeyError` exception.

We can also get the keyword value by indexing (a la Python lists):

```
>>> hdulist[0].header[27]
96
```

This example returns the 28th (like Python lists, it is 0-indexed) keyword’s value—an integer—96.

Similarly, it is easy to update a keyword’s value in Astropy, either through keyword name or index:

```
>>> prihdr = hdulist[0].header
>>> prihdr['targname'] = 'NGC121-a'
>>> prihdr[27] = 99
```

Please note however that almost all application code should update header values via their keyword name and not via their positional index. This is because most FITS keywords may appear at any position in the header.

It is also possible to update both the value and comment associated with a keyword by assigning them as a tuple:

```
>>> prihdr = hdulist[0].header
>>> prihdr['targname'] = ('NGC121-a', 'the observation target')
>>> prihdr['targname']
'NGC121-a'
>>> prihdr.comments['targname']
'the observation target'
```

Like a dict, one may also use the above syntax to add a new keyword/value pair (and optionally a comment as well). In this case the new card is appended to the end of the header (unless it's a commentary keyword such as COMMENT or HISTORY, in which case it is appended after the last card with that keyword).

Another way to either update an existing card or append a new one is to use the `Header.set()` method:

```
>>> prihdr.set('observer', 'Edwin Hubble')
```

Comment or history records are added like normal cards, though in their case a new card is always created, rather than updating an existing HISTORY or COMMENT card:

```
>>> prihdr['history'] = 'I updated this file 2/26/09'
>>> prihdr['comment'] = 'Edwin Hubble really knew his stuff'
>>> prihdr['comment'] = 'I like using HST observations'
>>> prihdr['history']
I updated this file 2/26/09
>>> prihdr['comment']
Edwin Hubble really knew his stuff
I like using HST observations
```

Note: Be careful not to confuse COMMENT cards with the comment value for normal cards.

To update existing COMMENT or HISTORY cards, reference them by index:

```
>>> prihdr['history'][0] = 'I updated this file on 2/27/09'
>>> prihdr['history']
I updated this file on 2/27/09
>>> prihdr['comment'][1] = 'I like using JWST observations'
>>> prihdr['comment']
Edwin Hubble really knew his stuff
I like using JWST observations
```

To see the entire header as it appears in the FITS file (with the END card and padding stripped), simply enter the header object by itself, or `print repr(header)`:

```
>>> header
SIMPLE = T / file does conform to FITS standard
BITPIX = 16 / number of bits per data pixel
NAXIS = 0 / number of data axes
all cards are shown...
>>> print repr(header)
identical...
```

Entering simply `print header` will also work, but may not be very legible on most displays, as this displays the header as it is written in the FITS file itself, which means there are no linebreaks between cards. This is a common

confusion in new users.

It's also possible to view a slice of the header:

```
>>> header[:2]
SIMPLE = T / file does conform to FITS standard
BITPIX = 16 / number of bits per data pixel
```

Only the first two cards are shown above.

To get a list of all keywords, use the `Header.keys()` method just as you would with a dict:

```
>>> prihdr.keys()
['SIMPLE', 'BITPIX', 'NAXIS', ...]
```

Working with Image Data

If an HDU's data is an image, the data attribute of the HDU object will return a numpy `ndarray` object. Refer to the numpy documentation for details on manipulating these numerical arrays.

```
>>> scidata = hdulist[1].data
```

Here, `scidata` points to the data object in the second HDU (the first HDU, `hdulist[0]`, being the primary HDU) which corresponds to the 'SCI' extension. Alternatively, you can access the extension by its extension name (specified in the EXTNAME keyword):

```
>>> scidata = hdulist['SCI'].data
```

If there is more than one extension with the same EXTNAME, the EXTVER value needs to be specified along with the EXTNAME as a tuple; e.g.:

```
>>> scidata = hdulist['sci',2].data
```

Note that the EXTNAME is also case-insensitive.

The returned numpy object has many attributes and methods for a user to get information about the array, e.g.

```
>>> scidata.shape
(800, 800)
>>> scidata.dtype.name
'float32'
```

Since image data is a numpy object, we can slice it, view it, and perform mathematical operations on it. To see the pixel value at `x=5, y=2`:

```
>>> print scidata[1, 4]
```

Note that, like C (and unlike FORTRAN), Python is 0-indexed and the indices have the slowest axis first and fastest changing axis last; i.e. for a 2-D image, the fast axis (X-axis) which corresponds to the FITS NAXIS1 keyword, is the second index. Similarly, the 1-indexed sub-section of `x=11 to 20` (inclusive) and `y=31 to 40` (inclusive) would be given in Python as:

```
>>> scidata[30:40, 10:20]
```

To update the value of a pixel or a sub-section:

```
>>> scidata[30:40, 10:20] = scidata[1, 4] = 999
```

This example changes the values of both the pixel `[1, 4]` and the sub-section `[30:40, 10:20]` to the new value of 999. See the [Numpy documentation](#) for more details on Python-style array indexing and slicing.

The next example of array manipulation is to convert the image data from counts to flux:

```
>>> photflam = hdulist[1].header['photflam']
>>> exptime = prihdr['exptime']
>>> scidata *= photflam / exptime
```

Note that performing an operation like this on an entire image requires holding the entire image in memory. This example performs the multiplication in-place so that no copies are made, but the original image must first be able to fit in main memory. For most observations this should not be an issue on modern personal computers.

If at this point you want to preserve all the changes you made and write it to a new file, you can use the `HDUList.writeto()` method (see below).

Working With Table Data

If you are familiar with numpy `recarray` (record array) objects, you will find the table data is basically a record array with some extra properties. But familiarity with record arrays is not a prerequisite for this guide.

Like images, the data portion of a FITS table extension is in the `.data` attribute:

```
>>> hdulist = fits.open('table.fits')
>>> tbdata = hdulist[1].data # assuming the first extension is a table
```

To see the first row of the table:

```
>>> print tbdata[0]
(1, 'abc', 3.7000002861022949, 0)
```

Each row in the table is a `FITS_record` object which looks like a (Python) tuple containing elements of heterogeneous data types. In this example: an integer, a string, a floating point number, and a Boolean value. So the table data are just an array of such records. More commonly, a user is likely to access the data in a column-wise way. This is accomplished by using the `field()` method. To get the first column (or “field” in Numpy parlance—it is used here interchangeably with “column”) of the table, use:

```
>>> tbdata.field(0)
array([1, 2])
```

A numpy object with the data type of the specified field is returned.

Like header keywords, a column can be referred either by index, as above, or by name:

```
>>> tbdata.field('id')
array([1, 2])
```

When accessing a column by name, dict-like access is also possible (and even preferable):

```
>>> tbdata['id']
array([1, 2])
```

In most cases it is preferable to access columns by their name, as the column name is entirely independent of its physical order in the table. As with header keywords, column names are case-insensitive.

But how do we know what columns we have in a table? First, let’s introduce another attribute of the table HDU: the `columns` attribute:

```
>>> cols = hdulist[1].columns
```

This attribute is a `ColDefs` (column definitions) object. If we use the `ColDefs.info()` method:

```
>>> cols.info()
name:
  ['c1', 'c2', 'c3', 'c4']
format:
  ['1J', '3A', '1E', '1L']
unit:
  ['', '', '', '']
null:
  [-2147483647, '', '', '']
bscale:
  ['', '', 3, '']
bzero:
  ['', '', 0.40000000000000002, '']
disp:
  ['I11', 'A3', 'G15.7', 'L6']
start:
  ['', '', '', '']
dim:
  ['', '', '', '']
```

it will show the attributes of all columns in the table, such as their names, formats, bscales, bzeros, etc. We can also get these properties individually; e.g.

```
>>> cols.names
['ID', 'name', 'mag', 'flag']
```

returns a (Python) list of field names.

Since each field is a Numpy object, we'll have the entire arsenal of Numpy tools to use. We can reassign (update) the values:

```
>>> tbdata['flag'][:] = 0
```

take the mean of a column:

```
>>> tbdata['mag'].mean()
>>> 84.4
```

and so on.

Save File Changes

As mentioned earlier, after a user opened a file, made a few changes to either header or data, the user can use `HDUList.writeto()` to save the changes. This takes the version of headers and data in memory and writes them to a new FITS file on disk. Subsequent operations can be performed to the data in memory and written out to yet another different file, all without recopying the original data to (more) memory.

```
>>> hdulist.writeto('newimage.fits')
```

will write the current content of `hdulist` to a new disk file `newfile.fits`. If a file was opened with the update mode, the `HDUList.flush()` method can also be used to write all the changes made since `open()`, back to the original file. The `close()` method will do the same for a FITS file opened with update mode:

```
>>> f = fits.open('original.fits', mode='update')
... # making changes in data and/or header
>>> f.flush() # changes are written back to original.fits
>>> f.close() # closing the file will also flush any changes and prevent
...         # further writing
```

14.2.2 Creating a New FITS File

Creating a New Image File

So far we have demonstrated how to read and update an existing FITS file. But how about creating a new FITS file from scratch? Such tasks are very easy in Astropy for an image HDU. We'll first demonstrate how to create a FITS file consisting only the primary HDU with image data.

First, we create a numpy object for the data part:

```
>>> import numpy as np
>>> n = np.arange(100.0) # a simple sequence of floats from 0.0 to 99.9
```

Next, we create a `PrimaryHDU` object to encapsulate the data:

```
>>> hdu = fits.PrimaryHDU(n)
```

We then create a `HDUList` to contain the newly created primary HDU, and write to a new file:

```
>>> hdulist = fits.HDUList([hdu])
>>> hdulist.writeto('new.fits')
```

That's it! In fact, Astropy even provides a shortcut for the last two lines to accomplish the same behavior:

```
>>> hdu.writeto('new.fits')
```

This will write a single HDU to a FITS file without having to manually encapsulate it in an `HDUList` object first.

Creating a New Table File

To create a table HDU is a little more involved than image HDU, because a table's structure needs more information. First of all, tables can only be an extension HDU, not a primary. There are two kinds of FITS table extensions: ASCII and binary. We'll use binary table examples here.

To create a table from scratch, we need to define columns first, by constructing the `Column` objects and their data. Suppose we have two columns, the first containing strings, and the second containing floating point numbers:

```
>>> from astropy.io import fits
>>> import numpy as np
>>> a1 = np.array(['NGC1001', 'NGC1002', 'NGC1003'])
>>> a2 = np.array([11.1, 12.3, 15.2])
>>> col1 = fits.Column(name='target', format='20A', array=a1)
>>> col2 = fits.Column(name='V_mag', format='E', array=a2)
```

Next, create a `ColDefs` (column-definitions) object for all columns:

```
>>> cols = fits.ColDefs([col1, col2])
```

Now, create a new binary table HDU object by using the `BinTableHDU.from_columns()` function:

```
>>> tbhdu = fits.BinTableHDU.from_columns(cols)
```

This function returns (in this case) a `BinTableHDU`.

Of course, you can do this more concisely without creating intermediate variables for the individual columns and without manually creating a `ColDefs` object:

```
>>> from astropy.io import fits
>>> tbhdu = fits.BinTableHDU.from_columns(
...     [fits.Column(name='target', format='20A', array=a1),
...     fits.Column(name='V_mag', format='E', array=a2)])
```

Now you may write this new table HDU directly to a FITS file like so:

```
>>> hdu = fits.PrimaryHDU(n)
```

This shortcut will automatically create a minimal primary HDU with no data and prepend it to the table HDU to create a valid FITS file. If you require additional data or header keywords in the primary HDU you may still create a `PrimaryHDU` object and build up the FITS file manually using an `HDUList`.

For example, first create a new `Header` object to encapsulate any keywords you want to include in the primary HDU, then as before create a `PrimaryHDU`:

```
>>> prihdr = fits.Header()
>>> prihdr['OBSERVER'] = 'Edwin Hubble'
>>> prihdr['COMMENT'] = "Here's some commentary about this FITS file."
>>> prihdu = fits.PrimaryHDU(header=prihdr)
```

When we create a new primary HDU with a custom header as in the above example, this will automatically include any additional header keywords that are *required* by the FITS format (keywords such as `SIMPLE` and `NAXIS` for example). In general, users should not have to manually manage such keywords, and should only create and modify observation-specific informational keywords.

We then create a `HDUList` containing both the primary HDU and the newly created table extension, and write to a new file:

```
>>> thdulist = fits.HDUList([prihdu, tbhdu])
>>> thdulist.writeto('table.fits')
```

Alternatively, we can append the table to the HDU list we already created in the image file section:

```
>>> hdulist.append(tbhdu)
>>> hdulist.writeto('image_and_table.fits')
```

The data structure used to represent FITS tables is called a `FITS_rec` and is derived from the `numpy.recarray` interface. When creating a new table HDU the individual column arrays will be assembled into a single `FITS_rec` array.

So far, we have covered the most basic features of `astropy.io.fits`. In the following chapters we'll show more advanced examples and explain options in each class and method.

14.2.3 Convenience Functions

`astropy.io.fits` also provides several high level (“convenience”) functions. Such a convenience function is a “canned” operation to achieve one simple task. By using these “convenience” functions, a user does not have to worry about opening or closing a file, all the housekeeping is done implicitly.

Warning: These functions are useful for interactive Python sessions and simple analysis scripts, but should not be used for application code, as they are highly inefficient. For example, each call to `getval()` requires re-parsing the entire FITS file. Code that makes repeated use of these functions should instead open the file with `open()` and access the data structures directly.

The first of these functions is `getheader()`, to get the header of an HDU. Here are several examples of getting the header. Only the file name is required for this function. The rest of the arguments are optional and flexible to specify

which HDU the user wants to access:

```
>>> from astropy.io.fits import getheader
>>> getheader('in.fits') # get default HDU (=0), i.e. primary HDU's header
>>> getheader('in.fits', 0) # get primary HDU's header
>>> getheader('in.fits', 2) # the second extension
>>> getheader('in.fits', 'sci') # the first HDU with EXTNAME='SCI'
>>> getheader('in.fits', 'sci', 2) # HDU with EXTNAME='SCI' and EXTVER=2
>>> getheader('in.fits', ('sci', 2)) # use a tuple to do the same
>>> getheader('in.fits', ext=2) # the second extension
>>> getheader('in.fits', extname='sci') # first HDU with EXTNAME='SCI'
>>> getheader('in.fits', extname='sci', extver=2)
```

Ambiguous specifications will raise an exception:

```
>>> getheader('in.fits', ext=('sci', 1), extname='err', extver=2)
...
TypeError: Redundant/conflicting extension arguments(s): {'ext': ('sci',
1), 'args': (), 'extver': 2, 'extname': 'err'}
```

After you get the header, you can access the information in it, such as getting and modifying a keyword value:

```
>>> from astropy.io.fits import getheader
>>> hdr = getheader('in.fits', 1) # get first extension's header
>>> filter = hdr['filter'] # get the value of the keyword "filter"
>>> val = hdr[10] # get the 11th keyword's value
>>> hdr['filter'] = 'FW555' # change the keyword value
```

For the header keywords, the header is like a dictionary, as well as a list. The user can access the keywords either by name or by numeric index, as explained earlier in this chapter.

If a user only needs to read one keyword, the `getval()` function can further simplify to just one call, instead of two as shown in the above examples:

```
>>> from astropy.io.fits import getval
>>> flt = getval('in.fits', 'filter', 1) # get 1st extension's keyword
# FILTER's value
>>> val = getval('in.fits', 10, 'sci', 2) # get the 2nd sci extension's
# 11th keyword's value
```

The function `getdata()` gets the data of an HDU. Similar to `getheader()`, it only requires the input FITS file name while the extension is specified through the optional arguments. It does have one extra optional argument `header`. If `header` is set to `True`, this function will return both data and header, otherwise only data is returned:

```
>>> from astropy.io.fits import getdata
>>> dat = getdata('in.fits', 'sci', 3) # get 3rd sci extension's data
... # get 1st extension's data and header
>>> data, hdr = getdata('in.fits', 1, header=True)
```

The functions introduced above are for reading. The next few functions demonstrate convenience functions for writing:

```
>>> fits.writeto('out.fits', data, header)
```

The `writeto()` function uses the provided data and an optional header to write to an output FITS file.

```
>>> fits.append('out.fits', data, header)
```

The `append()` function will use the provided data and the optional header to append to an existing FITS file. If the specified output file does not exist, it will create one.

```
>>> from astropy.io.fits import update
>>> update(file, dat, hdr, 'sci') # update the 'sci' extension
>>> update(file, dat, 3)         # update the 3rd extension
>>> update(file, dat, hdr, 3)    # update the 3rd extension
>>> update(file, dat, 'sci', 2)  # update the 2nd SCI extension
>>> update(file, dat, 3, header=hdr) # update the 3rd extension
>>> update(file, dat, header=hdr, ext=5) # update the 5th extension
```

The `update()` function will update the specified extension with the input data/header. The 3rd argument can be the header associated with the data. If the 3rd argument is not a header, it (and other positional arguments) are assumed to be the extension specification(s). Header and extension specs can also be keyword arguments.

Finally, the `info()` function will print out information of the specified FITS file:

```
>>> fits.info('test0.fits')
Filename: test0.fits
No. Name      Type          Cards Dimensions Format
0  PRIMARY PrimaryHDU   138  ()          Int16
1  SCI       ImageHDU      61  (400, 400) Int16
2  SCI       ImageHDU      61  (400, 400) Int16
3  SCI       ImageHDU      61  (400, 400) Int16
4  SCI       ImageHDU      61  (400, 400) Int16
```

This is one of the most useful convenience functions for getting an overview of what a given file contains without looking at any of the details.

14.3 Using `astropy.io.fits`

14.3.1 FITS Headers

In the next three chapters, more detailed information as well as examples will be explained for manipulating FITS headers, image/array data, and table data respectively.

Header of an HDU

Every HDU normally has two components: header and data. In Astropy these two components are accessed through the two attributes of the HDU, `hdu.header` and `hdu.data`.

While an HDU may have empty data, i.e. the `.data` attribute is `None`, any HDU will always have a header. When an HDU is created with a constructor, e.g. `hdu = PrimaryHDU(data, header)`, the user may supply the header value from an existing HDU's header and the data value from a numpy array. If the defaults (`None`) are used, the new HDU will have the minimal required keywords for an HDU of that type:

```
>>> hdu = fits.PrimaryHDU()
>>> hdu.header # show the all of the header cards
SIMPLE = T / conforms to FITS standard
BITPIX = 8 / array data type
NAXIS  = 0 / number of array dimensions
EXTEND = T
```

A user can use any header and any data to construct a new HDU. Astropy will strip any keywords that describe the data structure leaving only your informational keywords. Later it will add back in the required structural keywords for compatibility with the new HDU and any data added to it. So, a user can use a table HDU's header to construct an image HDU and vice versa. The constructor will also ensure the data type and dimension information in the header agree with the data.

The Header Attribute

Value Access, Updating, and Creating

As shown in the *Getting Started* tutorial, keyword values can be accessed via keyword name or index of an HDU's header attribute. Here is a quick summary:

```
>>> hdulist = fits.open('input.fits') # open a FITS file
>>> prihdr = hdulist[0].header # the primary HDU header
>>> print prihdr[3] # get the 4th keyword's value
10
>>> prihdr[3] = 20 # change its value
>>> prihdr['DARKCORR'] # get the value of the keyword 'darkcorr'
'OMIT'
>>> prihdr['darkcorr'] = 'PERFORM' # change darkcorr's value
```

Keyword names are case-insensitive except in a few special cases (see the sections on HIERARCH card and record-valued cards). Thus, `prihdr['abc']`, `prihdr['ABC']`, or `prihdr['aBc']` are all equivalent.

Like with Python's `dict` type, new keywords can also be added to the header using assignment syntax:

```
>>> 'DARKCORR' in header # Check for existence
False
>>> header['DARKCORR'] = 'OMIT' # Add a new DARKCORR keyword
```

You can also add a new value *and* comment by assigning them as a tuple:

```
>>> header['DARKCORR'] = ('OMIT', 'Dark Image Subtraction')
```

Note: An important point to note when adding new keywords to a header is that by default they are not appended *immediately* to the end of the file. Rather, they are appended to the last non-commentary keyword. This is in order to support the common use case of always having all HISTORY keywords grouped together at the end of a header. A new non-commentary keyword will be added at the end of the existing keywords, but before any HISTORY/COMMENT keywords at the end of the header.

There are a couple of ways to override this functionality:

- Use the `Header.append()` method with the `end=True` argument:

```
>>> header.append(('DARKCORR', 'OMIT', 'Dark Image Subtraction'),
                 end=True)
```

This forces the new keyword to be added at the actual end of the header.

- The `Header.insert()` method will always insert a new keyword exactly where you ask for it:

```
>>> header.insert(20, ('DARKCORR', 'OMIT', 'Dark Image Subtraction'))
```

This inserts the DARKCORR keyword before the 20th keyword in the header no matter what it is.

A keyword (and its corresponding card) can be deleted using the same index/name syntax:

```
>>> del prihdr[3] # delete the 2nd keyword
>>> del prihdr['abc'] # get the value of the keyword 'abc'
```

Note that, like a regular Python list, the indexing updates after each delete, so if `del prihdr[3]` is done two times in a row, the 4th and 5th keywords are removed from the original header. Likewise, `del prihdr[-1]` will delete the last card in the header.

It is also possible to delete an entire range of cards using the slice syntax:

```
>>> del prihdr[3:5]
```

The method `Header.set()` is another way to update the value or comment associated with an existing keyword, or to create a new keyword. Most of its functionality can be duplicated with the dict-like syntax shown above. But in some cases it might be more clear. It also has the advantage of allowing one to either move cards within the header, or specify the location of a new card relative to existing cards:

```
>>> prihdr.set('target', 'NGC1234', 'target name')
>>> # place the next new keyword before the 'TARGET' keyword
>>> prihdr.set('newkey', 666, before='TARGET') # comment is optional
>>> # place the next new keyword after the 21st keyword
>>> prihdr.set('newkey2', 42.0, 'another new key', after=20)
```

In FITS headers, each keyword may also have a comment associated with it explaining its purpose. The comments associated with each keyword are accessed through the `comments` attribute:

```
>>> header['NAXIS']
2
>>> header.comments['NAXIS']
the number of image axes
>>> header.comments['NAXIS'] = 'The number of image axes' # Update
```

Comments can be accessed in all the same ways that values are accessed, whether by keyword name or card index. Slices are also possible. The only difference is that you go through `header.comments` instead of just `header` by itself.

COMMENT, HISTORY, and Blank Keywords

Most keywords in a FITS header have unique names. If there are more than two cards sharing the same name, it is the first one accessed when referred by name. The duplicates can only be accessed by numeric indexing.

There are three special keywords (their associated cards are sometimes referred to as commentary cards), which commonly appear in FITS headers more than once. They are (1) blank keyword, (2) HISTORY, and (3) COMMENT. Unlike other keywords, when accessing these keywords they are returned as a list:

```
>>> prihdr['HISTORY']
I updated this file on 02/03/2011
I updated this file on 02/04/2011
....
```

These lists can be sliced like any other list. For example, to display just the last HISTORY entry, use `prihdr['history'][-1]`. Existing commentary cards can also be updated by using the appropriate index number for that card.

New commentary cards can be added like any other card by using the dict-like keyword assignment syntax, or by using the `Header.set()` method. However, unlike with other keywords, a new commentary card is always added and appended to the last commentary card with the same keyword, rather than to the end of the header. Here is an example:

```
>>> hdu.header['HISTORY'] = 'history 1'
>>> hdu.header[''] = 'blank 1'
>>> hdu.header['COMMENT'] = 'comment 1'
>>> hdu.header['HISTORY'] = 'history 2'
>>> hdu.header[''] = 'blank 2'
>>> hdu.header['COMMENT'] = 'comment 2'
```

and the part in the modified header becomes:

```
HISTORY history 1
HISTORY history 2
      blank 1
      blank 2
COMMENT comment 1
COMMENT comment 2
```

Users can also directly control exactly where in the header to add a new commentary card by using the `Header.insert()` method.

Note: Ironically, there is no comment in a commentary card, only a string value.

Card Images

A FITS header consists of card images.

A card image in a FITS header consists of a keyword name, a value, and optionally a comment. Physically, it takes 80 columns (bytes)—without carriage return—in a FITS file’s storage format. In Astropy, each card image is manifested by a `Card` object. There are also special kinds of cards: commentary cards (see above) and card images taking more than one 80-column card image. The latter will be discussed later.

Most of the time the details of dealing with cards are handled by the `Header` object, and it is not necessary to directly manipulate cards. In fact, most `Header` methods that accept a (keyword, value) or (keyword, value, comment) tuple as an argument can also take a `Card` object as an argument. `Card` objects are just wrappers around such tuples that provide the logic for parsing and formatting individual cards in a header. But there’s usually nothing gained by manually using a `Card` object, except to examine how a card might appear in a header before actually adding it to the header.

A new `Card` object is created with the `Card` constructor: `Card(key, value, comment)`. For example:

```
>>> c1 = fits.Card('TEMP', 80.0, 'temperature, floating value')
>>> c2 = fits.Card('DETECTOR', 1) # comment is optional
>>> c3 = fits.Card('MIR_REVR', True,
...               'mirror reversed? Boolean value')
>>> c4 = fits.Card('ABC', 2+3j, 'complex value')
>>> c5 = fits.Card('OBSERVER', 'Hubble', 'string value')

>>> print c1; print c2; print c3; print c4; print c5 # show the cards
TEMP = 80.0 / temperature, floating value
DETECTOR= 1 /
MIR_REVR= T / mirror reversed? Boolean value
ABC = (2.0, 3.0) / complex value
OBSERVER= 'Hubble ' / string value
```

Cards have the attributes `.keyword`, `.value`, and `.comment`. Both `.value` and `.comment` can be changed but not the `.keyword` attribute. In other words, once a card is created, it is created for a specific, immutable keyword.

The `Card()` constructor will check if the arguments given are conforming to the FITS standard and has a fixed card image format. If the user wants to create a card with a customized format or even a card which is not conforming to the FITS standard (e.g. for testing purposes), the `Card.fromstring()` class method can be used.

Cards can be verified with `Card.verify()`. The non-standard card `c2` in the example below is flagged by such verification. More about verification in Astropy will be discussed in a later chapter.

```
>>> c1 = fits.Card.fromstring('ABC = 3.456D023')
>>> c2 = fits.Card.fromstring("P.I. = 'Hubble'")
>>> print c1; print c2
ABC = 3.456D023
```


Examples follow:

```
>>> c = fits.Card('abcdefghi', 10)
Keyword name 'abcdefghi' is greater than 8 characters; a HIERARCH card will
be created.
>>> print c
HIERARCH abcdefghi = 10
>>> c = fits.Card('hierarch abcdefghi', 10)
>>> print c
HIERARCH abcdefghi = 10
>>> h = fits.PrimaryHDU()
>>> h.header['hierarch abcdefghi'] = 99
>>> h.header['abcdefghi']
99
>>> h.header['abcdefghi'] = 10
>>> h.header['abcdefghi']
10
>>> h.header['ABCDEFGHI']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "astropy/io/fits/header.py", line 121, in __getitem__
    return self._cards[self._cardindex(key)].value
  File "astropy/io/fits/header.py", line 1106, in _cardindex
    raise KeyError("Keyword %r not found." % keyword)
KeyError: "Keyword 'ABCDEFGHI.' not found."
>>> h.header
SIMPLE = T / conforms to FITS standard
BITPIX = 8 / array data type
NAXIS = 0 / number of array dimensions
EXTEND = T
HIERARCH abcdefghi = 1000
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "astropy/io/fits/header.py", line 121, in __getitem__
    return self._cards[self._cardindex(key)].value
  File "astropy/io/fits/header.py", line 1106, in _cardindex
    raise KeyError("Keyword %r not found." % keyword)
KeyError: "Keyword 'ABCDEFGHI.' not found."
```

Note: A final point to keep in mind about the `Header` class is that much of its design is intended to abstract away quirks about the FITS format. This is why, for example, it will automatically create `CONTINUE` and `HIERARCH` cards. The `Header` is just a data structure, and as a user you shouldn't have to worry about how it ultimately gets serialized to a header in a FITS file.

Though there are some areas where it's almost impossible to hide away the quirks of the FITS format, Astropy tries to make it so that you have to think about it as little as possible. If there are any areas where you have concern yourself unnecessarily about how the header is constructed, then let help@stsci.edu know, as there are probably areas where this can be improved on even more.

14.3.2 Image Data

In this chapter, we'll discuss the data component in an image HDU.

Image Data as an Array

A FITS primary HDU or an image extension HDU may contain image data. The following discussions apply to both of these HDU classes. In Astropy, for most cases, it is just a simple numpy array, having the shape specified by the NAXIS keywords and the data type specified by the BITPIX keyword - unless the data is scaled, see next section. Here is a quick cross reference between allowed BITPIX values in FITS images and the numpy data types:

BITPIX	Numpy Data Type
8	numpy.uint8 (note it is UNsigned integer)
16	numpy.int16
32	numpy.int32
-32	numpy.float32
-64	numpy.float64

To recap the fact that in numpy the arrays are 0-indexed and the axes are ordered from slow to fast. So, if a FITS image has NAXIS1=300 and NAXIS2=400, the numpy array of its data will have the shape of (400, 300).

Here is a summary of reading and updating image data values:

```
>>> f = fits.open('image.fits') # open a FITS file
>>> scidata = f[1].data # assume the first extension is an image
>>> print scidata[1,4] # get the pixel value at x=5, y=2
>>> scidata[30:40, 10:20] # get values of the subsection
... # from x=11 to 20, y=31 to 40 (inclusive)
>>> scidata[1,4] = 999 # update a pixel value
>>> scidata[30:40, 10:20] = 0 # update values of a subsection
>>> scidata[3] = scidata[2] # copy the 3rd row to the 4th row
```

Here are some more complicated examples by using the concept of the “mask array”. The first example is to change all negative pixel values in scidata to zero. The second one is to take logarithm of the pixel values which are positive:

```
>>> scidata[scidata < 0] = 0
>>> scidata[scidata > 0] = numpy.log(scidata[scidata > 0])
```

These examples show the concise nature of numpy array operations.

Scaled Data

Sometimes an image is scaled, i.e. the data stored in the file is not the image’s physical (true) values, but linearly transformed according to the equation:

$$\text{physical value} = \text{BSCALE} * (\text{storage value}) + \text{BZERO}$$

BSCALE and BZERO are stored as keywords of the same names in the header of the same HDU. The most common use of scaled image is to store unsigned 16-bit integer data because FITS standard does not allow it. In this case, the stored data is signed 16-bit integer (BITPIX=16) with BZERO=32768 (2^{15}), BSCALE=1.

Reading Scaled Image Data

Images are scaled only when either of the BSCALE/BZERO keywords are present in the header and either of their values is not the default value (BSCALE=1, BZERO=0).

For unscaled data, the data attribute of an HDU in Astropy is a numpy array of the same data type specified by the BITPIX keyword. For scaled image, the .data attribute will be the physical data, i.e. already transformed from the storage data and may not be the same data type as prescribed in BITPIX. This means an extra step of copying is needed and thus the corresponding memory requirement. This also means that the advantage of memory mapping is reduced for scaled data.

For floating point storage data, the scaled data will have the same data type. For integer data type, the scaled data will always be single precision floating point (`numpy.float32`). Here is an example of what happens to such a file, before and after the data is touched:

```
>>> f = fits.open('scaled_uint16.fits')
>>> hdu = f[1]
>>> print hdu.header['bitpix'], hdu.header['bzero']
16 32768
>>> print hdu.data # once data is touched, it is scaled
[ 11. 12. 13. 14. 15.]
>>> hdu.data.dtype.name
'float32'
>>> print hdu.header['bitpix'] # BITPIX is also updated
-32
>>> # BZERO and BSCALE are removed after the scaling
>>> print hdu.header['bzero']
KeyError: "Keyword 'bzero' not found."
```

Warning: An important caveat to be aware of when dealing with scaled data in PyFITS, is that when accessing the data via the `.data` attribute, the data is automatically scaled with the `BZERO` and `BSCALE` parameters. If the file was opened in “update” mode, it will be saved with the rescaled data. This surprising behavior is a compromise to err on the side of not losing data: If some floating point calculations were made on the data, rescaling it when saving could result in a loss of information.

To prevent this automatic scaling, open the file with the `do_not_scale_image_data=True` argument to `fits.open()`. This is especially useful for updating some header values, while ensuring that the data is not modified.

One may also manually reapply scale parameters by using `hdu.scale()` (see below). Alternately, one may open files with the `scale_back=True` argument. This assures that the original scaling is preserved when saving even when the physical values are updated. In other words, it reapplies the scaling to the new physical values upon saving.

Writing Scaled Image Data

With the extra processing and memory requirement, we discourage use of scaled data as much as possible. However, Astropy does provide ways to write scaled data with the `scale` method. Here are a few examples:

```
>>> # scale the data to Int16 with user specified bscale/bzero
>>> hdu.scale('int16', bzero=32768)
>>> # scale the data to Int32 with the min/max of the data range
>>> hdu.scale('int32', 'minmax')
>>> # scale the data, using the original BSCALE/BZERO
>>> hdu.scale('int32', 'old')
```

The first example above shows how to store an unsigned short integer array.

Great caution must be exercised when using the `scale()` method. The `data` attribute of an image HDU, after the `scale()` call, will become the storage values, not the physical values. So, only call `scale()` just before writing out to FITS files, i.e. calls of `writeto()`, `flush()`, or `close()`. No further use of the data should be exercised. Here is an example of what happens to the `data` attribute after the `scale()` call:

```
>>> hdu = fits.PrimaryHDU(numpy.array([0., 1, 2, 3]))
>>> print hdu.data
[ 0.  1.  2.  3.]
>>> hdu.scale('int16', bzero=32768)
>>> print hdu.data # now the data has storage values
```

```
[-32768 -32767 -32766 -32765]
>>> hdu.writeto('new.fits')
```

Data Sections

When a FITS image HDU's `data` is accessed, either the whole data is copied into memory (in cases of NOT using memory mapping or if the data is scaled) or a virtual memory space equivalent to the data size is allocated (in the case of memory mapping of non-scaled data). If there are several very large image HDUs being accessed at the same time, the system may run out of memory.

If a user does not need the entire image(s) at the same time, e.g. processing images(s) ten rows at a time, the `section` attribute of an HDU can be used to alleviate such memory problems.

With PyFITS' improved support for memory-mapping, the sections feature is not as necessary as it used to be for handling very large images. However, if the image's data is scaled with non-trivial BSCALE/BZERO values, accessing the data in sections may still be necessary under the current implementation. Memmap is also insufficient for loading images larger than 2 to 4 GB on a 32-bit system—in such cases it may be necessary to use sections.

Here is an example of getting the median image from 3 input images of the size 5000x5000:

```
>>> f1 = fits.open('file1.fits')
>>> f2 = fits.open('file2.fits')
>>> f3 = fits.open('file3.fits')
>>> output = numpy.zeros(5000 * 5000)
>>> for i in range(50):
...     j = i * 100
...     k = j + 100
...     x1 = f1[1].section[j:k,:]
...     x2 = f2[1].section[j:k,:]
...     x3 = f3[1].section[j:k,:]
...     # use scipy.stsci.image's median function
...     output[j:k] = image.median([x1, x2, x3])
```

Data in each `section` does not need to be contiguous for memory savings to be possible. PyFITS will do its best to join together discontinuous sections of the array while reading as little as possible into main memory.

Sections cannot currently be assigned to. Any modifications made to a data section are not saved back to the original file.

14.3.3 Table Data

In this chapter, we'll discuss the data component in a table HDU. A table will always be in an extension HDU, never in a primary HDU.

There are two kinds of table in the FITS standard: binary tables and ASCII tables. Binary tables are more economical in storage and faster in data access and manipulation. ASCII tables store the data in a "human readable" form and therefore take up more storage space as well as more processing time since the ASCII text needs to be parsed into numerical values.

Table Data as a Record Array

What is a Record Array?

A record array is an array which contains records (i.e. rows) of heterogeneous data types. Record arrays are available through the records module in the numpy library. Here is a simple example of record array:

```
>>> from numpy import rec
>>> bright = rec.array([(1, 'Sirius', -1.45, 'A1V'),
...                    (2, 'Canopus', -0.73, 'F0Ib'),
...                    (3, 'Rigel Kent', -0.1, 'G2V')],
...                   formats='int16,a20,float32,a10',
...                   names='order,name,mag,Sp')
```

In this example, there are 3 records (rows) and 4 fields (columns). The first field is a short integer, second a character string (of length 20), third a floating point number, and fourth a character string (of length 10). Each record has the same (heterogeneous) data structure.

The underlying data structure used for FITS tables is a class called `FITS_rec` which is a specialized subclass of `numpy.recarray`. A `FITS_rec` can be instantiated directly using the same initialization format presented for plain recarrays as in the example above. One may also instantiate a new `FITS_rec` from a list of PyFITS Column objects using the `FITS_rec.from_columns()` class method. This has the exact same semantics as `BinTableHDU.from_columns()` and `TableHDU.from_columns()`, except that it only returns an actual `FITS_rec` array and not a whole HDU object.

Metadata of a Table

The data in a FITS table HDU is basically a record array, with added attributes. The metadata, i.e. information about the table data, are stored in the header. For example, the keyword `TFORM1` contains the format of the first field, `TTYPE2` the name of the second field, etc. `NAXIS2` gives the number of records (rows) and `TFIELDS` gives the number of fields (columns). For FITS tables, the maximum number of fields is 999. The data type specified in `TFORM` is represented by letter codes for binary tables and a FORTRAN-like format string for ASCII tables. Note that this is different from the format specifications when constructing a record array.

Reading a FITS Table

Like images, the `.data` attribute of a table HDU contains the data of the table. To recap, the simple example in the Quick Tutorial:

```
>>> f = fits.open('bright_stars.fits') # open a FITS file
>>> tbdata = f[1].data # assume the first extension is a table
>>> print tbdata[:2] # show the first two rows
[(1, 'Sirius', -1.4500000476837158, 'A1V'),
 (2, 'Canopus', -0.73000001907348633, 'F0Ib')]

>>> print tbdata['mag'] # show the values in field "mag"
[-1.45000005 -0.73000002 -0.1 ]
>>> print tbdata.field(1) # columns can be referenced by index too
['Sirius' 'Canopus' 'Rigel Kent']
```

Note that in Astropy, when using the `field()` method, it is 0-indexed while the suffixes in header keywords, such as `TFORM` is 1-indexed. So, `tbdata.field(0)` is the data in the column with the name specified in `TTYPE1` and format in `TFORM1`.

Warning: The FITS format allows table columns with a zero-width data format, such as `'0D'`. This is probably intended as a space-saving measure on files in which that column contains no data. In such files, the zero-width columns are omitted when accessing the table data, so the indexes of fields might change when using the `field()` method. For this reason, if you expect to encounter files containing zero-width columns it is recommended to access fields by name rather than by index.

Table Operations

Selecting Records in a Table

Like image data, we can use the same “mask array” idea to pick out desired records from a table and make a new table out of it.

In the next example, assuming the table’s second field having the name ‘magnitude’, an output table containing all the records of magnitude > 5 from the input table is generated:

```
>>> from astropy.io import fits
>>> t = fits.open('table.fits')
>>> tbdata = t[1].data
>>> mask = tbdata['magnitude'] > 5
>>> newtbdata = tbdata[mask]
>>> hdu = fits.BinTableHDU(data=newtbdata)
>>> hdu.writeto('newtable.fits')
```

Merging Tables

Merging different tables is straightforward in Astropy. Simply merge the column definitions of the input tables:

```
>>> t1 = fits.open('table1.fits')
>>> t2 = fits.open('table2.fits')
>>> new_columns = t1[1].columns + t2[1].columns
>>> hdu = fits.BinTableHDU.from_columns(new_columns)
>>> hdu.writeto('newtable.fits')
```

The number of fields in the output table will be the sum of numbers of fields of the input tables. Users have to make sure the input tables don’t share any common field names. The number of records in the output table will be the largest number of records of all input tables. The expanded slots for the originally shorter table(s) will be zero (or blank) filled.

A simpler version of this example can be used to append a new column to a table. Updating an existing table with a new column is generally more difficult than it’s worth, but one can “append” a column to a table by creating a new table with columns from the existing table plus the new column(s):

```
>>> orig_table = fits.open('table.fits')[1].data
>>> orig_cols = orig_table.columns
>>> new_cols = fits.ColDefs([
...     fits.Column(name='NEWCOL1', format='D',
...                 array=np.zeros(len(orig_table))),
...     fits.Column(name='NEWCOL2', format='D',
...                 array=np.zeros(len(orig_table)))])
>>> hdu = fits.BinTableHDU.from_columns(orig_cols + new_cols)
>>> hdu.writeto('newtable.fits')
```

Now `newtable.fits` contains a new table with the original table, plus the two new columns filled with zeros.

Appending Tables

Appending one table after another is slightly trickier, since the two tables may have different field attributes. Here are two examples. The first is to append by field indices, the second one is to append by field names. In both cases, the output table will inherit column attributes (name, format, etc.) of the first table:

```

>>> t1 = fits.open('table1.fits')
>>> t2 = fits.open('table2.fits')
>>> nrow1 = t1[1].data.shape[0]
>>> nrow2 = t2[1].data.shape[0]
>>> nrow = nrow1 + nrow2
>>> hdu = fits.BinTableHDU.from_columns(t1[1].columns, nrow=nrow)
>>> for colname in t1[1].columns.names:
...     hdu.data[colname][nrow1:] = t2[1].data[colname]
>>> hdu.writeto('newtable.fits')

```

Scaled Data in Tables

A table field’s data, like an image, can also be scaled. Scaling in a table has a more generalized meaning than in images. In images, the physical data is a simple linear transformation from the storage data. The table fields do have such a construct too, where BSCALE and BZERO are stored in the header as TSCALn and TZEROn. In addition, boolean columns and ASCII tables’ numeric fields are also generalized “scaled” fields, but without TSCAL and TZERO.

All scaled fields, like the image case, will take extra memory space as well as processing. So, if high performance is desired, try to minimize the use of scaled fields.

All the scalings are done for the user, so the user only sees the physical data. Thus, this no need to worry about scaling back and forth between the physical and storage column values.

Creating a FITS Table

Column Creation

To create a table from scratch, it is necessary to create individual columns first. A `Column` constructor needs the minimal information of column name and format. Here is a summary of all allowed formats for a binary table:

FITS format code	Description	8-bit bytes
L	logical (Boolean)	1
X	bit	*
B	Unsigned byte	1
I	16-bit integer	2
J	32-bit integer	4
K	64-bit integer	4
A	character	1
E	single precision floating point	4
D	double precision floating point	8
C	single precision complex	8
M	double precision complex	16
P	array descriptor	8
Q	array descriptor	16

We’ll concentrate on binary tables in this chapter. ASCII tables will be discussed in a later chapter. The less frequently used X format (bit array) and P format (used in variable length tables) will also be discussed in a later chapter.

Besides the required name and format arguments in constructing a `Column`, there are many optional arguments which can be used in creating a column. Here is a list of these arguments and their corresponding header keywords and descriptions:

Argument in <code>Column()</code>	Corresponding header keyword	Description
--------------------------------------	---------------------------------	-------------

name	TTYPE	column name
format	TFORM	column format
unit	TUNIT	unit
null	TNULL	null value (only for B, I, and J)
bscale	TSCAL	scaling factor for data
bzero	TZERO	zero point for data scaling
disp	TDISP	display format
dim	TDIM	multi-dimensional array spec
start	TBCOL	starting position for ASCII table
array		the data of the column

Here are a few Columns using various combination of these arguments:

```
>>> import numpy as np
>>> from fits import Column
>>> counts = np.array([312, 334, 308, 317])
>>> names = np.array(['NGC1', 'NGC2', 'NGC3', 'NGC4'])
>>> c1 = Column(name='target', format='10A', array=names)
>>> c2 = Column(name='counts', format='J', unit='DN', array=counts)
>>> c3 = Column(name='notes', format='A10')
>>> c4 = Column(name='spectrum', format='1000E')
>>> c5 = Column(name='flag', format='L', array=[True, False, True, True])
```

In this example, formats are specified with the FITS letter codes. When there is a number (>1) preceding a (numeric type) letter code, it means each cell in that field is a one-dimensional array. In the case of column c4, each cell is an array (a numpy array) of 1000 elements.

For character string fields, the number be to the *left* of the letter 'A' when creating binary tables, and should be to the *right* when creating ASCII tables. However, as this is a common confusion both formats are understood when creating binary tables (note, however, that upon writing to a file the correct format will be written in the header). So, for columns c1 and c3, they both have 10 characters in each of their cells. For numeric data type, the dimension number must be before the letter code, not after.

After the columns are constructed, the `BinTableHDU.from_columns()` class method can be used to construct a table HDU. We can either go through the column definition object:

```
>>> coldefs = fits.ColDefs([c1, c2, c3, c4, c5])
>>> tbhdu = fits.BinTableHDU.from_columns(coldefs)
```

or directly use the `BinTableHDU.from_columns()` method:

```
>>> tbhdu = fits.BinTableHDU.from_columns([c1, c2, c3, c4, c5])
```

Note: Users familiar with older versions of PyFITS or Astropy will wonder what happened to `new_table()`. It is still there, but is deprecated. `BinTableHDU.from_columns()` and its companion for ASCII tables `TableHDU.from_columns()` are the same as `new_table()` in the arguments they accept and their behavior. They just make it more explicit what type of table HDU they create.

A look of the newly created HDU's header will show that relevant keywords are properly populated:

```
>>> tbhdu.header
XTENSION = 'BINTABLE' / binary table extension
BITPIX = 8 / array data type
NAXIS = 2 / number of array dimensions
NAXIS1 = 4025 / length of dimension 1
NAXIS2 = 4 / length of dimension 2
PCOUNT = 0 / number of group parameters
```

```

GCOUNT      =                               1 / number of groups
TFIELDS      =                               5 / number of table fields
TTYPER1      = 'target '
TFORM1       = '10A '
TTYPER2      = 'counts '
TFORM2       = 'J '
TUNIT2       = 'DN '
TTYPER3      = 'notes '
TFORM3       = '10A '
TTYPER4      = 'spectrum'
TFORM4       = '1000E '
TTYPER5      = 'flag '
TFORM5       = 'L '

```

Warning: It should be noted that when creating a new table with `BinTableHDU.from_columns()`, an in-memory copy of all of the input column arrays is created. This is because it is not guaranteed that the columns are arranged contiguously in memory in row-major order (in fact, they are most likely not), so they have to be combined into a new array.

However, if the array data *is* already contiguous in memory, such as in an existing record array, a kludge can be used to create a new table HDU without any copying. First, create the Columns as before, but without using the `array=` argument:

```
>>> c1 = Column(name='target', format='10A')
```

Then call `BinTableHDU.from_columns()`:

```
>>> tbhdu = fits.BinTableHDU.from_columns([c1, c2, c3, c4, c5])
```

This will create a new table HDU as before, with the correct column definitions, but an empty data section. Now simply assign your array directly to the HDU's data attribute:

```
>>> tbhdu.data = mydata
```

In a future version of Astropy table creation will be simplified and this process won't be necessary.

14.3.4 Verification

Astropy has built in a flexible scheme to verify FITS data being conforming to the FITS standard. The basic verification philosophy in Astropy is to be tolerant in input and strict in output.

When Astropy reads a FITS file which is not conforming to FITS standard, it will not raise an error and exit. It will try to make the best educated interpretation and only gives up when the offending data is accessed and no unambiguous interpretation can be reached.

On the other hand, when writing to an output FITS file, the content to be written must be strictly compliant to the FITS standard by default. This default behavior can be overwritten by several other options, so the user will not be held up because of a minor standard violation.

FITS Standard

Since FITS standard is a "loose" standard, there are many places the violation can occur and to enforce them all will be almost impossible. It is not uncommon for major observatories to generate data products which are not 100% FITS compliant. Some observatories have also developed their own sub-standard (dialect?) and some of these become so prevalent that they become de facto standards. Examples include the long string value and the use of the CONTINUE card.

The violation of the standard can happen at different levels of the data structure. Astropy's verification scheme is developed on these hierarchical levels. Here are the 3 Astropy verification levels:

1. The HDU List
2. Each HDU
3. Each Card in the HDU Header

These three levels correspond to the three categories of objects: `HDUList`, any HDU (e.g. `PrimaryHDU`, `ImageHDU`, etc.), and `Card`. They are the only objects having the `verify()` method. Most other classes in `astropy.io.fits` do not have a `verify()` method.

If `verify()` is called at the HDU List level, it verifies standard compliance at all three levels, but a call of `verify()` at the Card level will only check the compliance of that Card. Since Astropy is tolerant when reading a FITS file, no `verify()` is called on input. On output, `verify()` is called with the most restrictive option as the default.

Verification Options

There are several options accepted by all `verify(option)` calls in Astropy. In addition, they are available for the `output_verify` argument of the following methods: `close()`, `writeto()`, and `flush()`. In these cases, they are passed to a `verify()` call within these methods. The available options are:

exception

This option will raise an exception, if any FITS standard is violated. This is the default option for output (i.e. when `writeto()`, `close()`, or `flush()` is called. If a user wants to overwrite this default on output, the other options listed below can be used.

warn

This option is the same as the ignore option but will send warning messages. It will not try to fix any FITS standard violations whether fixable or not.

ignore

This option will ignore any FITS standard violation. On output, it will write the HDU List content to the output FITS file, whether or not it is conforming to the FITS standard.

The ignore option is useful in the following situations:

1. An input FITS file with non-standard formatting is read and the user wants to copy or write out to an output file. The non-standard formatting will be preserved in the output file.
2. A user wants to create a non-standard FITS file on purpose, possibly for testing or consistency.

No warning message will be printed out. This is like a silent warning option (see below).

fix

This option will try to fix any FITS standard violations. It is not always possible to fix such violations. In general, there are two kinds of FITS standard violations: fixable and non-fixable. For example, if a keyword has a floating number with an exponential notation in lower case 'e' (e.g. `1.23e11`) instead of the upper case 'E' as required by the FITS standard, it is a fixable violation. On the other hand, a keyword name like 'P.I.' is not fixable, since it will not know what to use to replace the disallowed periods. If a violation is fixable, this option will print out a message noting it is fixed. If it is not fixable, it will throw an exception.

The principle behind fixing is to do no harm. For example, it is plausible to 'fix' a Card with a keyword name like 'P.I.' by deleting it, but Astropy will not take such action to hurt the integrity of the data.

Not all fixes may be the "correct" fix, but at least Astropy will try to make the fix in such a way that it will not throw off other FITS readers.

silentfix

Same as `fix`, but will not print out informative messages. This may be useful in a large script where the user does not want excessive harmless messages. If the violation is not fixable, it will still throw an exception.

In addition, as of Astropy version 0.4.0 the following ‘combined’ options are available:

- **fix+ignore**
- **fix+warn**
- **fix+exception**
- **silentfix+ignore**
- **silentfix+warn**
- **silentfix+exception**

These options combine the semantics of the basic options. For example `silentfix+exception` is actually equivalent to just `silentfix` in that fixable errors will be fixed silently, but any unfixable errors will raise an exception. On the other hand `silentfix+warn` will issue warnings for unfixable errors, but will stay silent about any fixed errors.

Verifications at Different Data Object Levels

We’ll examine what Astropy’s verification does at the three different levels:

Verification at HDUList

At the HDU List level, the verification is only for two simple cases:

1. Verify that the first HDU in the HDU list is a Primary HDU. This is a fixable case. The fix is to insert a minimal Primary HDU into the HDU list.
2. Verify second or later HDU in the HDU list is not a Primary HDU. Violation will not be fixable.

Verification at Each HDU

For each HDU, the mandatory keywords, their locations in the header, and their values will be verified. Each FITS HDU has a fixed set of required keywords in a fixed order. For example, the Primary HDU’s header must at least have the following keywords:

```
SIMPLE =          T /
BITPIX =          8 /
NAXIS  =          0
```

If any of the mandatory keywords are missing or in the wrong order, the `fix` option will fix them:

```
>>> hdu.header          # has a 'bad' header
SIMPLE =                T /
NAXIS  =                0
BITPIX =                8 /
>>> hdu.verify('fix')  # fix it
Output verification result:
'BITPIX' card at the wrong place (card 2). Fixed by moving it to the right
place (card 1).
>>> h.header           # voila!
SIMPLE =                T / conforms to FITS standard
```

```
BITPIX = 8 / array data type
NAXIS = 0
```

Verification at Each Card

The lowest level, the Card, also has the most complicated verification possibilities. Here is a list of fixable and not fixable Cards:

Fixable Cards:

1. floating point numbers with lower case ‘e’ or ‘d’
2. the equal sign is before column 9 in the card image
3. string value without enclosing quotes
4. missing equal sign before column 9 in the card image
5. space between numbers and E or D in floating point values
6. unparseable values will be “fixed” as a string

Here are some examples of fixable cards:

```
>>> hdu.header[4:] # has a bunch of fixable cards
FIX1 = 2.1e23
FIX2= 2
FIX3 = string value without quotes
FIX4 2
FIX5 = 2.4 e 03
FIX6 = '2 10 '
>>> hdu.header[5] # can still access the values before the fix
2
>>> hdu.header['fix4']
2
>>> hdu.header['fix5']
2400.0
>>> hdu.verify('silentfix')
>>> hdu.header[4:]
FIX1 = 2.1E23
FIX2 = 2
FIX3 = 'string value without quotes'
FIX4 = 2
FIX5 = 2.4E03
FIX6 = '2 10 '
```

Unfixable Cards:

1. illegal characters in keyword name

We’ll summarize the verification with a “life-cycle” example:

```
>>> h = fits.PrimaryHDU() # create a PrimaryHDU
>>> # Try to add a non-standard FITS keyword 'P.I.' (FITS does not allow
>>> # '.' in the keyword), if using the update() method - doesn't work!
>>> h['P.I.'] = 'Hubble'
ValueError: Illegal keyword name 'P.I.'
>>> # Have to do it the hard way (so a user will not do this by accident)
>>> # First, create a card image and give verbatim card content (including
>>> # the proper spacing, but no need to add the trailing blanks)
>>> c = fits.Card.fromstring("P.I. = 'Hubble'")
```

```

>>> h.header.append(c) # then append it to the header
>>> # Now if we try to write to a FITS file, the default output
>>> # verification will not take it.
>>> h.writeto('pi.fits')
Output verification result:
HDU 0:
  Card 4:
    Unfixable error: Illegal keyword name 'P.I.'
.....
    raise VerifyError
VerifyError
>>> # Must set the output_verify argument to 'ignore', to force writing a
>>> # non-standard FITS file
>>> h.writeto('pi.fits', output_verify='ignore')
>>> # Now reading a non-standard FITS file
>>> # astropy.io.fits is magnanimous in reading non-standard FITS files
>>> hdus = fits.open('pi.fits')
>>> hdus[0].header
SIMPLE =          T / conforms to FITS standard
BITPIX =          8 / array data type
NAXIS  =          0 / number of array dimensions
EXTEND =          T
P.I.   = 'Hubble'
>>> # even when you try to access the offending keyword, it does NOT
>>> # complain
>>> hdus[0].header['p.i.'].
'Hubble'
>>> # But if you want to make sure if there is anything wrong/non-standard,
>>> # use the verify() method
>>> hdus.verify()
Output verification result:
HDU 0:
  Card 4:
    Unfixable error: Illegal keyword name 'P.I.'

```

Verification using the FITS Checksum Keyword Convention

The North American FITS committee has reviewed the FITS Checksum Keyword Convention for possible adoption as a FITS Standard. This convention provides an integrity check on information contained in FITS HDUs. The convention consists of two header keyword cards: CHECKSUM and DATASUM. The CHECKSUM keyword is defined as an ASCII character string whose value forces the 32-bit 1's complement checksum accumulated over all the 2880-byte FITS logical records in the HDU to equal negative zero. The DATASUM keyword is defined as a character string containing the unsigned integer value of the 32-bit 1's complement checksum of the data records in the HDU. Verifying the the accumulated checksum is still equal to negative zero provides a fairly reliable way to determine that the HDU has not been modified by subsequent data processing operations or corrupted while copying or storing the file on physical media.

In order to avoid any impact on performance, by default Astropy will not verify HDU checksums when a file is opened or generate checksum values when a file is written. In fact, CHECKSUM and DATASUM cards are automatically removed from HDU headers when a file is opened, and any CHECKSUM or DATASUM cards are stripped from headers when a HDU is written to a file. In order to verify the checksum values for HDUs when opening a file, the user must supply the checksum keyword argument in the call to the open convenience function with a value of True. When this is done, any checksum verification failure will cause a warning to be issued (via the warnings module). If checksum verification is requested in the open, and no CHECKSUM or DATASUM cards exist in the HDU header, the file will open without comment. Similarly, in order to output the CHECKSUM and DATASUM cards in an HDU header when writing to a file, the user must supply the checksum keyword argument with a value of True in the call to

the `writeto` function. It is possible to write only the DATASUM card to the header by supplying the `checksum` keyword argument with a value of `'datasum'`.

Here are some examples:

```
>>> # Open the file pix.fits verifying the checksum values for all HDUs
>>> hdul = fits.open('pix.fits', checksum=True)

>>> # Open the file in.fits where checksum verification fails for the
>>> # primary HDU
>>> hdul = fits.open('in.fits', checksum=True)
Warning: Checksum verification failed for HDU #0.

>>> # Create file out.fits containing an HDU constructed from data and
>>> # header containing both CHECKSUM and DATASUM cards.
>>> fits.writeto('out.fits', data, header, checksum=True)

>>> # Create file out.fits containing all the HDUs in the HDULIST
>>> # hdul with each HDU header containing only the DATASUM card
>>> hdul.writeto('out.fits', checksum='datasum')

>>> # Create file out.fits containing the HDU hdu with both CHECKSUM
>>> # and DATASUM cards in the header
>>> hdu.writeto('out.fits', checksum=True)

>>> # Append a new HDU constructed from array data to the end of
>>> # the file existingfile.fits with only the appended HDU
>>> # containing both CHECKSUM and DATASUM cards.
>>> fits.append('existingfile.fits', data, checksum=True)
```

14.3.5 Less Familiar Objects

In this chapter, we'll discuss less frequently used FITS data structures. They include ASCII tables, variable length tables, and random access group FITS files.

ASCII Tables

FITS standard supports both binary and ASCII tables. In ASCII tables, all the data are stored in a human readable text form, so it takes up more space and extra processing to parse the text for numeric data. Depending on how the columns are formatted, floating point data may also lose precision.

In Astropy, the interface for ASCII tables and binary tables is basically the same, i.e. the data is in the `.data` attribute and the `field()` method is used to refer to the columns and returns a numpy array. When reading the table, Astropy will automatically detect what kind of table it is.

```
>>> from astropy.io import fits
>>> hdus = fits.open('ascii_table.fits')
>>> hdus[1].data[:1]
FITS_rec(
... [(10.123000144958496, 37)],
... dtype=[('a', '>f4'), ('b', '>i4')])
>>> hdus[1].data['a']
array([ 10.12300014,  5.19999981, 15.60999966,  0. ,
 345. ], dtype=float32)
>>> hdus[1].data.formats
['E10.4', 'I5']
```

Note that the formats in the record array refer to the raw data which are ASCII strings (therefore ‘a11’ and ‘a5’), but the `.formats` attribute of data retains the original format specifications (‘E10.4’ and ‘I5’).

Creating an ASCII Table

Creating an ASCII table from scratch is similar to creating a binary table. The difference is in the Column definitions. The columns/fields in an ASCII table are more limited than in a binary table. It does not allow more than one numerical value in a cell. Also, it only supports a subset of what allowed in a binary table, namely character strings, integer, and (single and double precision) floating point numbers. Boolean and complex numbers are not allowed.

The format syntax (the values of the TFORM keywords) is different from that of a binary table, they are:

```
Aw          Character string
Iw          (Decimal) Integer
Fw.d       Single precision real
Ew.d       Single precision real, in exponential notation
Dw.d       Double precision real, in exponential notation
```

where, *w* is the width, and *d* the number of digits after the decimal point. The syntax difference between ASCII and binary tables can be confusing. For example, a field of 3-character string is specified ‘3A’ in a binary table and as ‘A3’ in an ASCII table.

The other difference is the need to specify the table type when using the `TableHDU.from_columns()` method, and that `Column` should be provided the `ascii=True` argument in order to be unambiguous.

Note: Although binary tables are more common in most FITS files, earlier versions of the FITS format only supported ASCII tables. That is why the class `TableHDU` is used for representing ASCII tables specifically, whereas `BinTableHDU` is more explicit that it represents a binary table. These names come from the value `XTENSION` keyword in the tables’ headers, which is `TABLE` for ASCII tables and `BINTABLE` for binary tables.

`TableHDU.from_columns()` can be used like so:

```
>>> import numpy as np
>>> from astropy.io import fits
>>> a1 = np.array(['abcd', 'def'])
>>> r1 = np.array([11., 12.])
>>> c1 = fits.Column(name='abc', format='A3', array=a1, ascii=True)
>>> c2 = fits.Column(name='def', format='E', array=r1, bscale=2.3,
...                 bzero=0.6, ascii=True)
>>> c3 = fits.Column(name='t1', format='I', array=[91, 92, 93],
...                 ascii=True)
>>> hdu = fits.TableHDU.from_columns([c1, c2, c3])
>>> hdu.writeto('ascii.fits')
>>> hdu.data
FITS_rec([( 'abcd', 11.0, 91), ('def', 12.0, 92), ('', 0.0, 93)],
         dtype=[('abc', '|S3'), ('def', '|S14'), ('t1', '|S10')])
```

It should be noted that when the formats of the columns are unambiguously specific to ASCII tables it is not necessary to specify `ascii=True` in the `ColDefs` constructor. In this case there *is* ambiguity because the format code ‘I’ represents a 16-bit integer in binary tables, while in ASCII tables it is not technically a valid format. ASCII table format codes technically require a character width for each column, such as ‘I10’ to create a column that can hold integers up to 10 characters wide.

However, PyFITS allows the width specification to be omitted in some cases. When it is omitted from ‘I’ format columns the minimum width needed to accurately represent all integers in the column is used. The only problem with using this shortcut is its ambiguity with the binary table ‘I’ format, so specifying `ascii=True` is a good practice (though PyFITS will still figure out what you meant in most cases).

Variable Length Array Tables

The FITS standard also supports variable length array tables. The basic idea is that sometimes it is desirable to have tables with cells in the same field (column) that have the same data type but have different lengths/dimensions. Compared with the standard table data structure, the variable length table can save storage space if there is a large dynamic range of data lengths in different cells.

A variable length array table can have one or more fields (columns) which are variable length. The rest of the fields (columns) in the same table can still be regular, fixed-length ones. Astropy will automatically detect what kind of field it is during reading; no special action is needed from the user. The data type specification (i.e. the value of the TFORM keyword) uses an extra letter ‘P’ and the format is

```
rPt (max)
```

where r is 0, 1, or absent, t is one of the letter code for regular table data type (L, B, X, I, J, etc. currently, the X format is not supported for variable length array field in Astropy), and max is the maximum number of elements. So, for a variable length field of int32, The corresponding format spec is, e.g. ‘PJ(100)’:

```
>>> f = fits.open('variable_length_table.fits')
>>> print f[1].header['tform5']
1PI(20)
>>> print f[1].data.field(4)[:3]
[array([1], dtype=int16) array([88, 2], dtype=int16)
 array([ 1, 88, 3], dtype=int16)]
```

The above example shows a variable length array field of data type int16 and its first row has one element, second row has 2 elements etc. Accessing variable length fields is almost identical to regular fields, except that operations on the whole field are usually not possible. A user has to process the field row by row.

Creating a Variable Length Array Table

Creating a variable length table is almost identical to creating a regular table. The only difference is in the creation of field definitions which are variable length arrays. First, the data type specification will need the ‘P’ letter, and secondly, the field data must be an objects array (as included in the numpy module). Here is an example of creating a table with two fields, one is regular and the other variable length array:

```
>>> from astropy.io import fits
>>> import numpy as np
>>> c1 = fits.Column(name='var', format='PJ()',
...                 array=np.array([[45., 56]
...                                 [11, 12, 13]]),
...                 dtype=np.object)
>>> c2 = fits.Column(name='xyz', format='2I', array=[[11, 3], [12, 4]])
>>> tbhdu = fits.BinTableHDU.from_columns([c1, c2])
>>> print tbhdu.data
FITS_rec([(array([45, 56]), array([11, 3], dtype=int16)),
          (array([11, 12, 13]), array([12, 4], dtype=int16))],
          dtype=[('var', '<i4', 2), ('xyz', '<i2', 2)])
>>> tbhdu.writeto('var_table.fits')
>>> hdu = fits.open('var_table.fits')
>>> hdu[1].header
XTENSION= 'BINTABLE'          / binary table extension
BITPIX   =                    8 / array data type
NAXIS    =                    2 / number of array dimensions
NAXIS1   =                   12 / length of dimension 1
NAXIS2   =                    2 / length of dimension 2
PCOUNT   =                   20 / number of group parameters
```

```

GCOUNT =          1 / number of groups
TFIELDS =          2 / number of table fields
TTYPER1 = 'var '
TFORM1 = 'PJ(3) '
TTYPER2 = 'xyz '
TFORM2 = '2I '

```

Random Access Groups

Another less familiar data structure supported by the FITS standard is the random access group. This convention was established before the binary table extension was introduced. In most cases its use can now be superseded by the binary table. It is mostly used in radio interferometry.

Like Primary HDUs, a Random Access Group HDU is always the first HDU of a FITS file. Its data has one or more groups. Each group may have any number (including 0) of parameters, together with an image. The parameters and the image have the same data type.

All groups in the same HDU have the same data structure, i.e. same data type (specified by the keyword BITPIX, as in image HDU), same number of parameters (specified by PCOUNT), and the same size and shape (specified by NAXISn keywords) of the image data. The number of groups is specified by GCOUNT and the keyword NAXIS1 is always 0. Thus the total data size for a Random Access Group HDU is

$$|\text{BITPIX}| * \text{GCOUNT} * (\text{PCOUNT} + \text{NAXIS2} * \text{NAXIS3} * \dots * \text{NAXISn})$$

Header and Summary

Accessing the header of a Random Access Group HDU is no different from any other HDU. Just use the `.header` attribute.

The content of the HDU can similarly be summarized by using the `HDUList.info()` method:

```

>>> f = fits.open('random_group.fits')
>>> print f[0].header['groups']
True
>>> print f[0].header['gcount']
7956
>>> print f[0].header['pcount']
6
>>> f.info()
Filename: random_group.fits
No. Name Type Cards Dimensions Format
0 AN GroupsHDU 158 (3, 4, 1, 1, 1) Float32 7956 Groups
6 Parameters

```

Data: Group Parameters

The data part of a random access group HDU is, like other HDUs, in the `.data` attribute. It includes both parameter(s) and image array(s).

Show the data in 100th group, including parameters and data:

```

>>> print f[0].data[99]
(-8.1987486677035799e-06, 1.2010923615889215e-05,
-1.011189139244005e-05, 258.0, 2445728., 0.10, array([[[[ 12.4308672 ,
0.56860745, 3.99993873],

```

```
[ 12.74043655, 0.31398511, 3.99993873],  
[ 0. , 0. , 3.99993873],  
[ 0. , 0. , 3.99993873]]]], dtype=float32))
```

The data first lists all the parameters, then the image array, for the specified group(s). As a reminder, the image data in this file has the shape of (1,1,1,4,3) in Python or C convention, or (3,4,1,1,1) in IRAF or FORTRAN convention.

To access the parameters, first find out what the parameter names are, with the `.parnames` attribute:

```
>>> f[0].data.parnames # get the parameter names  
['uu--', 'vv--', 'ww--', 'baseline', 'date', 'date']
```

The group parameter can be accessed by the `par()` method. Like the table `field()` method, the argument can be either index or name:

```
>>> print f[0].data.par(0)[99] # Access group parameter by name or by index  
-8.1987486677035799e-06  
>>> print f[0].data.par('uu--')[99]  
-8.1987486677035799e-06
```

Note that the parameter name ‘date’ appears twice. This is a feature in the random access group, and it means to add the values together. Thus:

```
>>> f[0].data.parnames # get the parameter names  
['uu--', 'vv--', 'ww--', 'baseline', 'date', 'date']  
>>> print f[0].data.par(4)[99] # Duplicate parameter name 'date'  
2445728.0  
>>> print f[0].data.par(5)[99]  
0.10  
>>> # When accessed by name, it adds the values together if the name is  
>>> # shared by more than one parameter  
>>> print f[0].data.par('date')[99]  
2445728.10
```

The `par()` is a method for either the entire data object or one data item (a group). So there are two possible ways to get a group parameter for a certain group, this is similar to the situation in table data (with its `field()` method):

```
>>> print f[0].data.par(0)[99]  
-8.1987486677035799e-06  
>>> print f[0].data[99].par(0)  
-8.1987486677035799e-06
```

On the other hand, to modify a group parameter, we can either assign the new value directly (if accessing the row/group number last) or use the `setpar()` method (if accessing the row/group number first). The method `setpar()` is also needed for updating by name if the parameter is shared by more than one parameters:

```
>>> # Update group parameter when selecting the row (group) number last  
>>> f[0].data.par(0)[99] = 99.  
>>> # Update group parameter when selecting the row (group) number first  
>>> f[0].data[99].setpar(0, 99.) # or setpar('uu--', 99.)  
>>>  
>>> # Update group parameter by name when the name is shared by more than  
>>> # one parameters, the new value must be a tuple of constants or  
>>> # sequences  
>>> f[0].data[99].setpar('date', (2445729., 0.3))  
>>> f[0].data[:3].setpar('date', (2445729., [0.11, 0.22, 0.33]))  
>>> f[0].data[:3].par('date')  
array([ 2445729.11 , 2445729.22 , 2445729.33000001])
```

Data: Image Data

The image array of the data portion is accessible by the `data` attribute of the data object. A numpy array is returned:

```
>>> print f[0].data.data[99]
array([[[[[ 12.4308672 , 0.56860745, 3.99993873],
 [ 12.74043655, 0.31398511, 3.99993873],
 [ 0. , 0. , 3.99993873],
 [ 0. , 0. , 3.99993873]]]], type=float32)
```

Creating a Random Access Group HDU

To create a random access group HDU from scratch, use `GroupData` to encapsulate the data into the group data structure, and use `GroupsHDU` to create the HDU itself:

```
>>> # Create the image arrays. The first dimension is the number of groups.
>>> imdata = numpy.arange(100.0, shape=(10, 1, 1, 2, 5))
>>> # Next, create the group parameter data, we'll have two parameters.
>>> # Note that the size of each parameter's data is also the number of
>>> # groups.
>>> # A parameter's data can also be a numeric constant.
>>> pdata1 = numpy.arange(10) + 0.1
>>> pdata2 = 42
>>> # Create the group data object, put parameter names and parameter data
>>> # in lists assigned to their corresponding arguments.
>>> # If the data type (bitpix) is not specified, the data type of the
>>> # image will be used.
>>> x = fits.GroupData(imdata, parnames=['abc', 'xyz'],
...                   padata=[pdata1, pdata2], bitpix=-32)
>>> # Now, create the GroupsHDU and write to a FITS file.
>>> hdu = fits.GroupsHDU(x)
>>> hdu.writeto('test_group.fits')
>>> hdu.header
SIMPLE =          T / conforms to FITS standard
BITPIX =        -32 / array data type
NAXIS   =          5 / number of array dimensions
NAXIS1  =          0
NAXIS2  =          5
NAXIS3  =          2
NAXIS4  =          1
NAXIS5  =          1
EXTEND  =          T
GROUPS  =          T / has groups
PCOUNT  =          2 / number of parameters
GCOUNT  =         10 / number of groups
PTYPE1  = 'abc '
PTYPE2  = 'xyz '
>>> print hdu.data[:2]
FITS_rec[
(0.10000000149011612, 42.0, array([[[[ 0., 1., 2., 3., 4.],
 [ 5., 6., 7., 8., 9.]]]], dtype=float32)),
(1.1000000238418579, 42.0, array([[[[ 10., 11., 12., 13., 14.],
 [ 15., 16., 17., 18., 19.]]]], dtype=float32))
]
```

Compressed Image Data

A general technique has been developed for storing compressed image data in FITS binary tables. The principle used in this convention is to first divide the n-dimensional image into a rectangular grid of sub images or ‘tiles’. Each tile is then compressed as a continuous block of data, and the resulting compressed byte stream is stored in a row of a variable length column in a FITS binary table. Several commonly used algorithms for compressing image tiles are supported. These include, Gzip, Rice, IRAF Pixel List (PLIO), and Hcompress.

For more details, reference “A FITS Image Compression Proposal” from:

<http://www.adass.org/adass/proceedings/adass99/P2-42/>

and “Registered FITS Convention, Tiled Image Compression Convention”:

<http://fits.gsfc.nasa.gov/registry/tilecompression.html>

Compressed image data is accessed, in Astropy, using the optional “`astropy.io.fits.compression`” module contained in a C shared library (`compression.so`). If an attempt is made to access an HDU containing compressed image data when the compression module is not available, the user is notified of the problem and the HDU is treated like a standard binary table HDU. This notification will only be made the first time compressed image data is encountered. In this way, the compression module is not required in order for Astropy to work.

Header and Summary

In Astropy, the header of a compressed image HDU appears to the user like any image header. The actual header stored in the FITS file is that of a binary table HDU with a set of special keywords, defined by the convention, to describe the structure of the compressed image. The conversion between binary table HDU header and image HDU header is all performed behind the scenes. Since the HDU is actually a binary table, it may not appear as a primary HDU in a FITS file.

The content of the HDU header may be accessed using the `.header` attribute:

```
>>> f = fits.open('compressed_image.fits')
>>> print f[1].header
XTENSION= 'IMAGE'           / extension type
BITPIX   =                  16 / array data type
NAXIS    =                   2 / number of array dimensions
NAXIS1   =                 512 / length of data axis
NAXIS2   =                 512 / length of data axis
PCOUNT   =                   0 / number of parameters
GCOUNT   =                   1 / one data group (required keyword)
EXTNAME  = 'COMPRESSED'     / name of this binary table extension
```

The contents of the corresponding binary table HDU may be accessed using the hidden `._header` attribute. However, all user interface with the HDU header should be accomplished through the image header (the `.header` attribute):

```
>>> f = fits.open('compressed_image.fits')
>>> print f[1]._header
XTENSION= 'BINTABLE'       / binary table extension
BITPIX   =                   8 / 8-bit bytes
NAXIS    =                   2 / 2-dimensional binary table
NAXIS1   =                   8 / width of table in bytes
NAXIS2   =                 512 / number of rows in table
PCOUNT   =             157260 / size of special data area
GCOUNT   =                   1 / one data group (required keyword)
TFIELDS  =                   1 / number of fields in each row
TTYPE1   = 'COMPRESSED_DATA' / label for field 1
TFORM1   = '1PB(384)'      / data format of field: variable length array
ZIMAGE   =                  T / extension contains compressed image
```

```

ZBITPIX =          16 / data type of original image
ZNAXIS  =           2 / dimension of original image
ZNAXIS1 =          512 / length of original image axis
ZNAXIS2 =          512 / length of original image axis
ZTILE1  =          512 / size of tiles to be compressed
ZTILE2  =           1 / size of tiles to be compressed
ZCMTYPE= 'RICE_1  '   / compression algorithm
ZNAME1  = 'BLOCKSIZE' / compression block size
ZVAL1   =           32 / pixels per block
EXTNAME = 'COMPRESSED' / name of this binary table extension

```

The contents of the HDU can be summarized by using either the `info()` convenience function or method:

```

>>> fits.info('compressed_image.fits')
Filename: compressed_image.fits
No.    Name          Type          Cards   Dimensions   Format
0     PRIMARY       PrimaryHDU    6      ()          int16
1     COMPRESSED   CompImageHDU  52     (512, 512)  int16
>>>
>>> f = fits.open('compressed_image.fits')
>>> f.info()
Filename: compressed_image.fits
No.    Name          Type          Cards   Dimensions   Format
0     PRIMARY       PrimaryHDU    6      ()          int16
1     COMPRESSED   CompImageHDU  52     (512, 512)  int16
>>>

```

Data

As with the header, the data of a compressed image HDU appears to the user as standard uncompressed image data. The actual data is stored in the fits file as Binary Table data containing at least one column (`COMPRESSED_DATA`). Each row of this variable-length column contains the byte stream that was generated as a result of compressing the corresponding image tile. Several optional columns may also appear. These include, `UNCOMPRESSED_DATA` to hold the uncompressed pixel values for tiles that cannot be compressed, `ZSCALE` and `ZZERO` to hold the linear scale factor and zero point offset which may be needed to transform the raw uncompressed values back to the original image pixel values, and `ZBLANK` to hold the integer value used to represent undefined pixels (if any) in the image.

The contents of the uncompressed HDU data may be accessed using the `.data` attribute:

```

>>> f = fits.open('compressed_image.fits')
>>> f[1].data
array([[38, 43, 35, ..., 45, 43, 41],
       [36, 41, 37, ..., 42, 41, 39],
       [38, 45, 37, ..., 42, 35, 43],
       ...,
       [49, 52, 49, ..., 41, 35, 39],
       [57, 52, 49, ..., 40, 41, 43],
       [53, 57, 57, ..., 39, 35, 45]], dtype=int16)

```

The compressed data can be accessed via the `.compressed_data` attribute, but this rarely need be accessed directly. It may be useful for performing direct copies of the compressed data without needing to decompress it first.

Creating a Compressed Image HDU

To create a compressed image HDU from scratch, simply construct a `CompImageHDU` object from an uncompressed image data array and its associated image header. From there, the HDU can be treated just like any other image HDU:

```
>>> hdu = fits.CompImageHDU(imageData, imageHeader)
>>> hdu.writeto('compressed_image.fits')
```

The API documentation for the `CompImageHDU` initializer method describes the possible options for constructing a `CompImageHDU` object.

14.3.6 Executable Scripts

Astropy installs a couple of useful utility programs on your system that are built with Astropy.

fitsheader

`fitsheader` is a command line script based on `astropy.io.fits` for printing the header(s) of one or more FITS file(s) to the standard output in a human-readable format.

Example uses of `fitsheader`:

1. Print the header of all the HDUs of a .fits file:

```
$ fitsheader filename.fits
```

2. Print the header of the third HDU extension:

```
$ fitsheader --ext 3 filename.fits
```

3. Print the header of a named extension, e.g. to select the HDU with header keywords `EXTNAME='SCI'` and `EXTVER='2'`:

```
$ fitsheader --ext "SCI,2" filename.fits
```

4. Print the headers of all fits files in a directory:

```
$ fitsheader *.fits
```

Note that compressed images (HDUs of type `CompImageHDU`) really have two headers: a real `BINTABLE` header to describe the compressed data, and a fake `IMAGE` header representing the image that was compressed. Astropy returns the latter by default. You must supply the `--compressed` option if you require the real header that describes the compression.

With Astropy installed, please run `fitsheader --help` to see the full usage documentation.

fitscheck

`fitscheck` is a command line script based on `astropy.io.fits` for verifying and updating the `CHECKSUM` and `DATA-SUM` keywords of .fits files. `fitscheck` can also detect and often fix other FITS standards violations. `fitscheck` facilitates re-writing the non-standard checksums originally generated by `astropy.io.fits` with standard checksums which will interoperate with `CFITSIO`.

`fitscheck` will refuse to write new checksums if the checksum keywords are missing or their values are bad. Use `--force` to write new checksums regardless of whether or not they currently exist or pass. Use `--ignore-missing` to tolerate missing checksum keywords without comment.

Example uses of `fitscheck`:

1. Verify and update checksums, tolerating non-standard checksums, updating to standard checksum:

```
$ fitscheck --checksum either --write *.fits
```

2. Write new checksums, even if existing checksums are bad or missing:

```
$ fitscheck --write --force *.fits
```

3. Verify standard checksums and FITS compliance without changing the files:

```
$ fitscheck --compliance *.fits
```

4. Verify original nonstandard checksums only:

```
$ fitscheck --checksum nonstandard *.fits
```

5. Only check and fix compliance problems, ignoring checksums:

```
$ fitscheck --checksum none --compliance --write *.fits
```

6. Verify standard interoperable checksums:

```
$ fitscheck *.fits
```

7. Delete checksum keywords:

```
$ fitscheck --checksum none --write *.fits
```

With Astropy installed, please run `fitscheck --help` to see the full program usage documentation.

fitsdiff

`fitsdiff` provides a thin command-line wrapper around the `FITSDiff` interface—it outputs the report from a `FITSDiff` of two FITS files, and like common diff-like commands returns a 0 status code if no differences were found, and 1 if differences were found:

With Astropy installed, please run `fitscheck --help` to see the full program usage documentation.

14.3.7 Miscellaneous Features

This section describes some of the miscellaneous features of `astropy.io.fits`.

Differs

The `astropy.io.fits.diff` module contains several facilities for generating and reporting the differences between two FITS files, or two components of a FITS file.

The `FITSDiff` class can be used to generate and represent the differences between either two FITS files on disk, or two existing `HDUList` objects (or some combination thereof).

Likewise, the `HeaderDiff` class can be used to find the differences just between two `Header` objects. Other available differs include `HDUDiff`, `ImageDataDiff`, `TableDataDiff`, and `RawDataDiff`.

Each of these classes are instantiated with two instances of the objects that they diff. The returned diff instance has a number of attributes starting with `.diff_` that describe differences between the two objects.

For example the `HeaderDiff` class can be used to find the differences between two `Header` objects like so:

```
>>> from astropy.io import fits
>>> header1 = fits.Header([('KEY_A', 1), ('KEY_B', 2)])
>>> header2 = fits.Header([('KEY_A', 3), ('KEY_C', 4)])
>>> diff = fits.diff.HeaderDiff(header1, header2)
>>> diff.identical
False
>>> diff.diff_keywords
(['KEY_B'], ['KEY_C'])
>>> diff.diff_keyword_values
defaultdict(<function <lambda> at ...>, {'KEY_A': [(1, 3)]})
```

See the API documentation for details on the different differ classes.

14.3.8 Examples

Converting a 3-color image (JPG) to separate FITS images



Figure 14.1: Red color information

```
#!/usr/bin/env python
import numpy
```



Figure 14.2: Green color information



Figure 14.3: Blue color information

```
import Image

from astropy.io import fits

# get the image and color information
image = Image.open('hs-2009-14-a-web.jpg')
# image.show()
xsize, ysize = image.size
r, g, b = image.split()
rdata = r.getdata() # data is now an array of length ysize*xsize
gdata = g.getdata()
bdata = b.getdata()

# create numpy arrays
npr = numpy.reshape(rdata, (ysize, xsize))
npg = numpy.reshape(gdata, (ysize, xsize))
npb = numpy.reshape(bdata, (ysize, xsize))

# write out the fits images, the data numbers are still JUST the RGB
# scalings; don't use for science
red = fits.PrimaryHDU(data=npr)
red.header['LATOBS'] = "32:11:56" # add spurious header info
red.header['LONGOBS'] = "110:56"
red.writeto('red.fits')

green = fits.PrimaryHDU(data=npg)
green.header['LATOBS'] = "32:11:56"
green.header['LONGOBS'] = "110:56"
green.writeto('green.fits')
```

```
blue = fits.PrimaryHDU(data=npb)
blue.header['LATOBS'] = "32:11:56"
blue.header['LONGOBS'] = "110:56"
blue.writeto('blue.fits')
```

14.4 Other Information

14.4.1 astropy.io.fits FAQ

Contents

- [astropy.io.fits FAQ](#)
 - [General Questions](#)
 - * [What is PyFITS and how does it relate to Astropy?](#)
 - * [What is the development status of PyFITS?](#)
 - [Usage Questions](#)
 - * [Something didn't work as I expected. Did I do something wrong?](#)
 - * [Astropy crashed and output a long string of code. What do I do?](#)
 - * [Why does opening a file work in CFITSIO, ds9, etc. but not in Astropy?](#)
 - * [How do I turn off the warning messages Astropy keeps outputting to my console?](#)
 - * [What convention does Astropy use for indexing, such as of image coordinates?](#)
 - * [How do I open a very large image that won't fit in memory?](#)
 - * [How can I create a very large FITS file from scratch?](#)
 - * [How do I create a multi-extension FITS file from scratch?](#)
 - * [Why is an image containing integer data being converted unexpectedly to floats?](#)
 - * [Why am I losing precision when I assign floating point values in the header?](#)
 - * [Why is reading rows out of a FITS table so slow?](#)
 - [Comparison with Other FITS Readers](#)
 - * [What is the difference between astropy.io.fits and fitsio?](#)
 - * [Why did Astropy adopt PyFITS as its FITS reader instead of fitsio?](#)
 - * [What performance differences are there between astropy.io.fits and fitsio?](#)
 - * [Why is fitsio so much faster than Astropy at reading tables?](#)

General Questions

What is PyFITS and how does it relate to Astropy?

PyFITS is a library written in, and for use with the Python programming language for reading, writing, and manipulating FITS formatted files. It includes a high-level interface to FITS headers with the ability for high and low-level manipulation of headers, and it supports reading image and table data as Numpy arrays. It also supports more obscure and non-standard formats found in some FITS files.

The `astropy.io.fits` module is identical to PyFITS but with the names changed. When development began on Astropy it was clear that one of the core requirements would be a FITS reader. Rather than starting from scratch, PyFITS—being the most flexible FITS reader available for Python—was ported into Astropy. There are plans to gradually phase out PyFITS as a stand-alone module and deprecate it in favor of `astropy.io.fits`. See more about that in the next question.

Although PyFITS is written mostly in Python, it includes an optional module written in C that's required to read/write compressed image data. However, the rest of PyFITS functions without this extension module.

What is the development status of PyFITS?

PyFITS is written and maintained by the Science Software Branch at the [Space Telescope Science Institute](#), and is licensed by [AURA](#) under a [3-clause BSD license](#) (see [LICENSE.txt](#) in the PyFITS source code).

It is now primarily developed as primarily as a component of Astropy (`astropy.io.fits`) rather than as a stand-alone module. There are a few reasons for this: The first is simply to reduce development effort; the overhead of maintaining both PyFITS *and* `astropy.io.fits` in separate code bases is non-trivial. The second is that there are many features of Astropy (units, tables, etc.) from which the `astropy.io.fits` module can benefit greatly. Since PyFITS is already integrated into Astropy, it makes more sense to continue development there rather than make Astropy a dependency of PyFITS.

PyFITS' current primary developer and active maintainer is [Erik Bray](#), though patch submissions are welcome from anyone. PyFITS is now primarily developed in a Git repository for ease of merging to and from Astropy. Patches and issue reports can be posted to the [GitHub project](#) for PyFITS, or for Astropy. There is also a legacy [Trac site](#) with some older issue reports still open, but new issues should be submitted via GitHub if possible. An [SVN mirror](#) of the repository is still maintained as well.

The current stable release series is 3.3.x. Each 3.3.x release tries to contain only bug fixes, and to not introduce any significant behavioral or API changes (though this isn't guaranteed to be perfect). Patch releases for older release series may be released upon request. Older versions of PyFITS (2.4 and earlier) are no longer actively supported.

Usage Questions

Something didn't work as I expected. Did I do something wrong?

Possibly. But if you followed the documentation and things still did not work as expected, it is entirely possible that there is a mistake in the documentation, a bug in the code, or both. So feel free to report it as a bug. There are also many, many corner cases in FITS files, with new ones discovered almost every week. `astropy.io.fits` is always improving, but does not support all cases perfectly. There are some features of the FITS format (scaled data, for example) that are difficult to support correctly and can sometimes cause unexpected behavior.

For the most common cases, however, such as reading and updating FITS headers, images, and tables, `astropy.io.fits` is very stable and well-tested. Before every Astropy/PyFITS release it is ensured that all its tests pass on a variety of platforms, and those tests cover the majority of use-cases (until new corner cases are discovered).

Astropy crashed and output a long string of code. What do I do?

This listing of code is what is known as a [stack trace](#) (or in Python parlance a "traceback"). When an unhandled exception occurs in the code, causing the program to end, this is a way of displaying where the exception occurred and the path through the code that led to it.

As Astropy is meant to be used as a piece in other software projects, some exceptions raised by Astropy are by design. For example, one of the most common exceptions is a `KeyError` when an attempt is made to read the value of a non-existent keyword in a header:

```
>>> from astropy.io import fits
>>> h = fits.Header()
>>> h['NAXIS']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/path/to/astropy/io/fits/header.py", line 125, in __getitem__
    return self._cards[self._cardindex(key)].value
  File "/path/to/astropy/io/fits/header.py", line 1535, in _cardindex
```

```
    raise KeyError("Keyword %r not found." % keyword)
KeyError: "Keyword 'NAXIS' not found."
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/path/to/astropy/io/fits/header.py", line 125, in __getitem__
    return self._cards[self._cardindex(key)].value
  File "/path/to/astropy/io/fits/header.py", line 1535, in _cardindex
    raise KeyError("Keyword %r not found." % keyword)
KeyError: "Keyword 'NAXIS' not found."
```

This indicates that something was looking for a keyword called “NAXIS” that does not exist. If an error like this occurs in some other software that uses Astropy, it may indicate a bug in that software, in that it expected to find a keyword that didn’t exist in a file.

Most “expected” exceptions will output a message at the end of the traceback giving some idea of why the exception occurred and what to do about it. The more vague and mysterious the error message in an exception appears, the more likely that it was caused by a bug in Astropy. So if you’re getting an exception and you really don’t know why or what to do about it, feel free to report it as a bug.

Why does opening a file work in CFITSIO, ds9, etc. but not in Astropy?

As mentioned elsewhere in this FAQ, there are many unusual corner cases when dealing with FITS files. It’s possible that a file should work, but isn’t support due to a bug. Sometimes it’s even possible for a file to work in an older version of Astropy or PyFITS, but not a newer version due to a regression that isn’t tested for yet.

Another problem with the FITS format is that, as old as it is, there are many conventions that appear in files from certain sources that do not meet the FITS standard. And yet they are so common-place that it is necessary to support them in any FITS readers. CONTINUE cards are one such example. There are non-standard conventions supported by Astropy/PyFITS that are not supported by CFITSIO and possibly vice-versa. You may have hit one of those cases.

If Astropy is having trouble opening a file, a good way to rule out whether not the problem is with Astropy is to run the file through the `fitsverify` program. For smaller files you can even use the [online FITS verifier](#). These use CFITSIO under the hood, and should give a good indication of whether or not there is something erroneous about the file. If the file is malformed, fitsverify will output errors and warnings.

If fitsverify confirms no problems with a file, and Astropy is still having trouble opening it (especially if it produces a traceback) then it’s possible there is a bug in Astropy.

How do I turn off the warning messages Astropy keeps outputting to my console?

Astropy uses Python’s built-in `warnings` subsystem for informing about exceptional conditions in the code that are recoverable, but that the user may want to be informed of. One of the most common warnings in `astropy.io.fits` occurs when updating a header value in such a way that the comment must be truncated to preserve space:

```
Card is too long, comment is truncated.
```

Any console output generated by Astropy can be assumed to be from the warnings subsystem. See Astropy’s documentation on the *Python warnings system* for more information on how to control and quiet warnings.

What convention does Astropy use for indexing, such as of image coordinates?

All arrays and sequences in Astropy use a zero-based indexing scheme. For example, the first keyword in a header is `header[0]`, not `header[1]`. This is in accordance with Python itself, as well as C, on which Python is based.

This may come as a surprise to veteran FITS users coming from IRAF, where 1-based indexing is typically used, due to its origins in FORTRAN.

Likewise, the top-left pixel in an $N \times N$ array is `data[0, 0]`. The indices for 2-dimensional arrays are row-major order, in that the first index is the row number, and the second index is the column number. Or put in terms of axes, the first axis is the y-axis, and the second axis is the x-axis. This is the opposite of column-major order, which is used by FORTRAN and hence FITS. For example, the second index refers to the axis specified by NAXIS1 in the FITS header.

In general, for N-dimensional arrays, row-major orders means that the right-most axis is the one that varies the fastest while moving over the array data linearly. For example, the 3-dimensional array:

```
[[[1, 2],
  [3, 4]],
 [[5, 6],
  [7, 8]]]
```

is represented linearly in row-major order as:

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

Since 2 immediately follows 1, you can see that the right-most (or inner-most) axis is the one that varies the fastest.

The discrepancy in axis-ordering may take some getting used to, but it is a necessary evil. Since most other Python and C software assumes row-major ordering, trying to enforce column-major ordering in arrays returned by Astropy is likely to cause more difficulties than it's worth.

How do I open a very large image that won't fit in memory?

In PyFITS, prior to version 3.1, when the data portion of an HDU is accessed, the data is read into memory in its entirety. For example:

```
>>> hdul = pyfits.open('myimage.fits')
>>> hdul[0].data
...
```

reads the entire image array from disk into memory. For very large images or tables this is clearly undesirable, if not impossible given the available resources.

However, `astropy.io.fits.open` has an option to access the data portion of an HDU by memory mapping using `mmap`. In both Astropy and newer versions of PyFITS this is used by *default*.

What this means is that accessing the data as in the example above only reads portions of the data into memory on demand. For example, if I request just a slice of the image, such as `hdul[0].data[100:200]`, then just rows 100-200 will be read into memory. This happens transparently, as though the entire image were already in memory. This works the same way for tables. For most cases this is your best bet for working with large files.

To ensure use of memory mapping, just add the `memmap=True` argument to `fits.open`. Likewise, using `memmap=False` will force data to be read entirely into memory.

The default can also be controlled through a configuration option called `USE_MEMMAP`. Setting this to 0 will disable `mmap` by default.

Unfortunately, memory mapping does not currently work as well with scaled image data, where BSCALE and BZERO factors need to be applied to the data to yield physical values. Currently this requires enough memory to hold the entire array, though this is an area that will see improvement in the future.

An alternative, which currently only works for image data (that is, non-tables) is the sections interface. It is largely replaced by the better support for `mmap`, but may still be useful on systems with more limited virtual-memory space,

such as on 32-bit systems. Support for scaled image data is flakey with sections too, though that will be fixed. See the documentation on *image sections* for more details on using this interface.

How can I create a very large FITS file from scratch?

This is a very common issue, but unfortunately Astropy does not come with any built-in facilities for creating large files (larger than will fit in memory) from scratch (though it may in the future).

Normally to create a single image FITS file one would do something like:

```
>>> import numpy
>>> from astropy.io import fits
>> data = numpy.zeros((40000, 40000), dtype=numpy.float64)
>> hdu = fits.PrimaryHDU(data=data)
>> hdu.writeto('large.fits')
```

However, a 40000 x 40000 array of doubles is nearly twelve gigabytes! Most systems won't be able to create that in memory just to write out to disk. In order to create such a large file efficiently requires a little extra work, and a few assumptions.

First, it is helpful to anticipate about how large (as in, how many keywords) the header will have in it. FITS headers must be written in 2880 byte blocks—large enough for 36 keywords per block (including the END keyword in the final block). Typical headers have somewhere between 1 and 4 blocks, though sometimes more.

Since the first thing we write to a FITS file is the header, we want to write enough header blocks so that there is plenty of padding in which to add new keywords without having to resize the whole file. Say you want the header to use 4 blocks by default. Then, excluding the END card which Astropy will add automatically, create the header and pad it out to $36 * 4$ cards like so:

```
>>> data = numpy.zeros((100, 100), dtype=numpy.float64)
# This is a stub array that we'll be using to initialize the HDU; its
# exact size is irrelevant, as long as it has the desired number of
# dimensions
>>> hdu = fits.PrimaryHDU(data=data)
>>> header = hdu.header
>>> while len(header) < (36 * 4 - 1):
...     header.append() # Adds a blank card to the end
```

Now adjust the NAXISn keywords to the desired size of the array, and write *only* the header out to a file. Using the `hdu.writeto()` method will cause Astropy to “helpfully” reset the NAXISn keywords to match the size of the dummy array. That is because it works hard to ensure that only valid FITS files are written. Instead, we can write *just* the header to a file using the `Header.tofile` method:

```
>>> header['NAXIS1'] = 40000
>>> header['NAXIS2'] = 40000
>>> header.tofile('large.fits')
```

Finally, we need to grow out the end of the file to match the length of the data (plus the length of the header). This can be done very efficiently on most systems by seeking past the end of the file and writing a single byte, like so:

```
>>> with open('large.fits', 'rb+') as fobj:
...     # Seek past the length of the header, plus the length of the
...     # Data we want to write.
...     # The -1 is to account for the final byte that we are about to
...     # write:
...     fobj.seek(len(header.tostring()) + (40000 * 40000 * 8) - 1)
...     fobj.write('\0')
```

On modern operating systems this will cause the file (past the header) to be filled with zeros out to the ~12GB needed to hold a 40000 x 40000 image. On filesystems that support sparse file creation (most Linux filesystems, but not the HFS+ filesystem used by most Macs) this is a very fast, efficient operation. On other systems your mileage may vary.

This isn't the only way to build up a large file, but probably one of the safest. This method can also be used to create large multi-extension FITS files, with a little care.

For creating very large tables, this method may also be used. Though it can be difficult to determine ahead of time how many rows a table will need. In general, use of the `astropy.io.fits` module is currently discouraged for the creation and manipulation of large tables. The FITS format itself is not designed for efficient on-disk or in-memory manipulation of table structures. For large, heavy-duty table data it might be better to look into using [HDF5](#) through the [PyTables](#) library. The *Astropy Table* interface can provide an abstraction layer between different on-disk table formats as well (for example for converting a table between FITS and HDF5).

PyTables makes use of Numpy under the hood, and can be used to write binary table data to disk in the same format required by FITS. It is then possible to serialize your table to the FITS format for distribution. At some point this FAQ might provide an example of how to do this.

How do I create a multi-extension FITS file from scratch?

When you open a FITS file with `astropy.io.fits.open`, an `HDUList` object is returned, which holds all the HDUs in the file. This `HDUList` class is a subclass of Python's builtin `list`, and can be created from scratch and used as such:

```
>>> from astropy.io import fits
>>> new_hdul = fits.HDUList()
>>> new_hdul.append(fits.ImageHDU())
>>> new_hdul.append(fits.ImageHDU())
>>> new_hdul.writeto('test.fits')
```

Or the HDU instances can be created first (or read from an existing FITS file) and the `HDUList` instantiated like so:

```
>>> hdu1 = fits.PrimaryHDU()
>>> hdu2 = fits.ImageHDU()
>>> new_hdul = fits.HDUList([hdu1, hdu2])
>>> new_hdul.writeto('test.fits')
```

That will create a new multi-extension FITS file with two empty IMAGE extensions (a default PRIMARY HDU is prepended automatically if one was not provided manually).

Why is an image containing integer data being converted unexpectedly to floats?

If the header for your image contains non-trivial values for the optional `BSCALE` and/or `BZERO` keywords (that is, `BSCALE != 1` and/or `BZERO != 0`), then the raw data in the file must be rescaled to its physical values according to the formula:

$$\text{physical_value} = \text{BZERO} + \text{BSCALE} * \text{array_value}$$

As `BZERO` and `BSCALE` are floating point values, the resulting value must be a float as well. If the original values were 16-bit integers, the resulting values are single-precision (32-bit) floats. If the original values were 32-bit integers the resulting values are double-precision (64-bit floats).

This automatic scaling can easily catch you off guard if you're not expecting it, because it doesn't happen until the data portion of the HDU is accessed (to allow things like updating the header without rescaling the data). For example:

```
>>> hdul = fits.open('scaled.fits')
>>> image = hdul['SCI', 1]
>>> image.header['BITPIX']
32
>>> image.header['BSCALE']
2.0
>>> data = image.data # Read the data into memory
>>> data.dtype
dtype('float64') # Got float64 despite BITPIX = 32 (32-bit int)
>>> image.header['BITPIX'] # The BITPIX will automatically update too
-64
>>> 'BSCALE' in image.header # And the BSCALE keyword removed
False
```

The reason for this is that once a user accesses the data they may also manipulate it and perform calculations on it. If the data were forced to remain as integers, a great deal of precision is lost. So it is best to err on the side of not losing data, at the cost of causing some confusion at first.

If the data must be returned to integers before saving, use the `ImageHDU.scale` method:

```
>>> image.scale('int32')
>>> image.header['BITPIX']
32
```

Alternatively, if a file is opened with `mode='update'` along with the `scale_back=True` argument, the original BSCALE and BZERO scaling will be automatically re-applied to the data before saving. Usually this is not desirable, especially when converting from floating point back to unsigned integer values. But this may be useful in cases where the raw data needs to be modified corresponding to changes in the physical values.

To prevent rescaling from occurring at all (good for updating headers—even if you don't intend for the code to access the data, it's good to err on the side of caution here), use the `do_not_scale_image_data` argument when opening the file:

```
>>> hdul = fits.open('scaled.fits', do_not_scale_image_data=True)
>>> image = hdul['SCI', 1]
>>> image.data.dtype
dtype('int32')
```

Why am I losing precision when I assign floating point values in the header?

The FITS standard allows two formats for storing floating-point numbers in a header value. The “fixed” format requires the ASCII representation of the number to be in bytes 11 through 30 of the header card, and to be right-justified. This leaves a standard number of characters for any comment string.

The fixed format is not wide enough to represent the full range of values that can be stored in a 64-bit float with full precision. So FITS also supports a “free” format in which the ASCII representation can be stored anywhere, using the full 70 bytes of the card (after the keyword).

Currently Astropy/PyFITS only supports writing fixed format (it can read both formats), so all floating point values assigned to a header are stored in the fixed format. There are plans to add support for more flexible formatting.

In the meantime it is possible to add or update cards by manually formatting the card image from a string, as it should appear in the FITS file:

```
>>> c = fits.Card.fromstring('FOO          = 1234567890.123456789')
>>> h = fits.Header()
>>> h.append(c)
```

```
>>> h
FOO      = 1234567890.123456789
```

As long as you don't assign new values to 'FOO' via `h['FOO'] = 123`, will maintain the header value exactly as you formatted it (as long as it is valid according to the FITS standard).

Why is reading rows out of a FITS table so slow?

Underlying every table data array returned by `astropy.io.fits` is a Numpy `recarray` which is a Numpy array type specifically for representing structured array data (i.e. a table). As with normal image arrays, Astropy accesses the underlying binary data from the FITS file via mmap (see the question "[What performance differences are there between astropy.io.fits and fitsio?](#)" for a deeper explanation for this). The underlying mmap is then exposed as a `recarray` and in general this is a very efficient way to read the data.

However, for many (if not most) FITS tables it isn't all that simple. For many columns there are conversions that have to take place between the actual data that's "on disk" (in the FITS file) and the data values that are returned to the user. For example FITS binary tables represent boolean values differently from how Numpy expects them to be represented, "Logical" columns need to be converted on the fly to a format Numpy (and hence the user) can understand. This issue also applies to data that is linearly scaled via the `TSCALn` and `TZEROn` header keywords.

Supporting all of these "FITS-isms" introduces a lot of overhead that might not be necessary for all tables, but are still common nonetheless. That's not to say it can't be faster even while supporting the peculiarities of FITS-CFITSIO for example supports all the same features but is orders of magnitude faster. Astropy could do much better here too, and there are many known issues causing slowdown. There are plenty of opportunities for speedups, and patches are welcome. In the meantime for high-performance applications with FITS tables some users might find the `fitsio` library more to their liking.

Comparison with Other FITS Readers

What is the difference between `astropy.io.fits` and `fitsio`?

The `astropy.io.fits` module (originally PyFITS) is a "pure Python" FITS reader in that all the code for parsing the FITS file format is in Python, though Numpy is used to provide access to the FITS data via the `ndarray` interface. `astropy.io.fits` currently also accesses the `CFITSIO` to support the FITS Tile Compression convention, but this feature is optional. It does not use `CFITSIO` outside of reading compressed images.

`fitsio`, on the other hand, is a Python wrapper for the `CFITSIO` library. All the heavy lifting of reading the FITS format is handled by `CFITSIO`, while `fitsio` provides an easier to use object-oriented API including providing a Numpy interface to FITS files read from `CFITSIO`. Much of it is written in C (to provide the interface between Python and `CFITSIO`), and the rest is in Python. The Python end mostly provides the documentation and user-level API.

Because `fitsio` wraps `CFITSIO` it inherits most of its strengths and weaknesses, though it has an added strength of providing an easier to use API than if one were to use `CFITSIO` directly.

Why did Astropy adopt PyFITS as its FITS reader instead of `fitsio`?

When the Astropy project was first started it was clear from the start that one of its core components should be a sub-module for reading and writing FITS files, as many other components would be likely to depend on this functionality. At the time, the `fitsio` package was in its infancy (it goes back to roughly 2011) while PyFITS had already been established going back to before the year 2000). It was already a mature package with support for the vast majority of FITS files found in the wild, including outdated formats such as "Random Groups" FITS files still used extensively in the radio astronomy community.

Although many aspects of PyFITS' interface have evolved over the years, much of it has also remained the same, and is already familiar to astronomers working with FITS files in Python. Most of not all existing training materials were also based around PyFITS. PyFITS was developed at STScI, which also put forward significant resources to develop Astropy, with an eye toward integrating Astropy into STScI's own software stacks. As most of the Python software at STScI uses PyFITS it was the only practical choice for making that transition.

Finally, although CFITSIO (and by extension `fitsio`) can read any FITS files that conform to the FITS standard, it does not support all of the non-standard conventions that have been added to FITS files in the wild. It does have some support for some of these conventions (such as CONTINUE cards and, to a limited extent, HIERARCH cards), it is not easy to add support for other conventions to a large and complex C codebase.

PyFITS' object-oriented design makes supporting non-standard conventions somewhat easier in most cases, and as such PyFITS can be more flexible in the types of FITS files it can read and return *useful* data from. This includes better support for files that fail to meet the FITS standard, but still contain useful data that should still be readable at least well-enough to correct any violations of the FITS standard. For example, a common error in non-English-speaking regions is to insert non-ASCII characters into FITS headers. This is not a valid FITS file, but still should be readable in some sense. Supporting structural errors such as this is more difficult in CFITSIO which assumes a more rigid structure.

What performance differences are there between `astropy.io.fits` and `fitsio`?

There are two main performance areas to look at: reading/parsing FITS headers and reading FITS data (image-like arrays as well as tables).

In the area of headers `fitsio` is significantly faster in most cases. This is due in large part to the (almost) pure C implementation (due to the use of CFITSIO), but also due to fact that it is more rigid and does not support as many local conventions and other special cases as `astropy.io.fits` tries to support in its pure Python implementation.

That said the difference is small, and only likely to be a bottleneck either when opening files containing thousands of HDUs, or reading the headers out of thousands of FITS files in succession (in either case the difference is not even an order of magnitude).

Where data is concerned the situation is a little more complicated, and requires some understanding of how PyFITS is implemented versus CFITSIO and `fitsio`. First it's important to understand how they differ in terms of memory management.

`astropy.io.fits`/PyFITS uses `mmap`, by default, to provide access to the raw binary data in FITS files. `Mmap` is a system call (or in most cases these days a wrapper in your `libc` for a lower-level system call) which allows user-space applications to essentially do the same thing your OS is doing when it uses a pagefile (swap space) for virtual memory: It allows data in a file on disk to be paged into physical memory one page (or in practice usually several pages) at a time on an as-needed basis. These cached pages of the file are also accessible from all processes on the system, so multiple processes can read from the same file with little additional overhead. In the case of reading over all the data in the file the performance difference between using `mmap` versus reading the entire data into physical memory at once can vary widely between systems, hardware, and depending on what else is happening on the system at the moment, but `mmap` almost always going to be better.

In principle it requires more overhead since accessing each page will result in a page fault, and the system requires more requests to the disk. But in practice the OS will optimize this pretty aggressively, especially for the most common case of sequential access—also in reality reading the entire thing into memory is still going to result in a whole lot of page faults too. For random access having all the data in physical memory is always going to be best, though with `mmap` it's usually going to be pretty good too (one doesn't normally access all the data in a file in totally random order—usually a few sections of it will be accessed most frequently, the OS will keep those pages in physical memory as best it can). So for the most general case of reading FITS files (or most large data on disk) this is the best choice, especially for casual users, and is hence enabled by default.

CFITSIO/`fitsio`, on the other hand, doesn't assume the existence of technologies like `mmap` and page caching. Thus it implements its own LRU cache of I/O buffers that store sections of FITS files read from disk in memory in

FITS' famous 2880 byte chunk size. The I/O buffers are used heavily in particular for keeping the headers in memory. Though for large data reads (for example reading an entire image from a file) it *does* bypass the cache and instead does a read directly from disk into a user-provided memory buffer.

However, even when CFITSIO reads direct from the file, this is still largely less efficient than using mmap: Normally when your OS reads a file from disk, it caches as much of that read as it can in physical memory (in its page cache) so that subsequent access to those same pages does not require a subsequent expensive disk read. This happens when using mmap too, since the data has to be copied from disk into RAM at some point. The difference is that when using mmap to access the data, the program is able to read that data *directly* out of the OS's page cache (so long as it's only being read). On the other hand when reading data from a file into a local buffer such as with `fread()`, the data is first read into the page cache (if not already present) and then copied from the page cache into the local buffer. So every read performs at least one additional memory copy per page read (requiring twice as much physical memory, and possibly lots of paging if the file is large and pages need to be dropped from the cache).

The user API for CFITSIO usually works by having the user allocate a memory buffer large enough to hold the image/table they want to read (or at least the section they're interested in). There are some helper functions for determining the appropriate amount of space to allocate. Then you just pass it a pointer to your buffer and CFITSIO handles all the reading (usually using the process described above), and copies the results into your user buffer. For large reads it reads directly from the file into your buffer. Though if the data needs to be scaled it makes a stop in CFITSIO's own buffer first, then writes the rescaled values out to the user buffer (if rescaling has been requested). Regardless, this means that if your program wishes to hold an entire image in memory at once it will use as much RAM as the size of the data. For most applications it's better (and sufficient) to write it work on smaller sections of the data, but this requires extra complexity. Using mmap on the other hand makes managing this complexity simpler and more efficient.

A very simple and informal test demonstrates this difference. This test was performed on four simple FITS images (one of which is a cube) of dimensions 256x256, 1024x1024, 4096x4096, and 256x1024x1024. Each image was generated before the test and filled with randomized 64-bit floating point values. A similar test was performed using both `astropy.io.fits` and `fitsio`: A handle to the FITS file is opened using each library's basic semantics, and then the entire data array of the files is copied into a temporary array in memory (for example if we were blitting the image to a video buffer). For Astropy the test is written:

```
def read_test_pyfits(filename):
    with fits.open(filename, memmap=True) as hdul:
        data = hdul[0].data
        c = data.copy()
```

The test was timed in IPython on a Linux system with kernel version 2.6.32, a 6-core Intel Xeon X5650 CPU clocked at 2.67 GHz per core, and 11.6 GB of RAM using:

```
for filename in filenames:
    print(filename)
    %timeit read_test_pyfits(filename)
```

where `filenames` is just a list of the aforementioned generated sample files. The results were:

```
256x256.fits
1000 loops, best of 3: 1.28 ms per loop
1024x1024.fits
100 loops, best of 3: 4.24 ms per loop
4096x4096.fits
10 loops, best of 3: 60.6 ms per loop
256x1024x1024.fits
1 loops, best of 3: 1.15 s per loop
```

For `fitsio` the test was:

```
def read_test_fitsio(filename):
    with fitsio.FITS(filename) as f:
        data = f[0].read()
        c = data.copy()
```

This was also run in a loop over all the sample files, producing the results:

```
256x256.fits
1000 loops, best of 3: 476 µs per loop
1024x1024.fits
100 loops, best of 3: 12.2 ms per loop
4096x4096.fits
10 loops, best of 3: 136 ms per loop
256x1024x1024.fits
1 loops, best of 3: 3.65 s per loop
```

It should be made clear that the sample files were rewritten with new random data between the Astropy test and the fitsio test, so they were not reading the same data from the OS's page cache. Fitsio was much faster on the small (256x256) image because in that case the time is dominated by parsing the headers. As already explained this is much faster in CFITSIO. However, as the data size goes up and the header parsing no longer dominates the time, `astropy.io.fits` using mmap is roughly twice as fast. This discrepancy would be almost entirely due to it requiring roughly half as many in-memory copies to read the data, as explained earlier. That said, more extensive benchmarking could be very interesting.

This is also not to say that `astropy.io.fits` does better in all cases. There are some cases where it is currently blown away by fitsio. See the subsequent question.

Why is fitsio so much faster than Astropy at reading tables?

In many cases it isn't—there is either no difference, or it may be a little faster in Astropy depending on what you're trying to do with the table and what types of columns or how many columns the table has. There are some cases, however, where fitsio can be radically faster, mostly for reasons explained above in “[Why is reading rows out of a FITS table so slow?](#)”

In principle a table is no different from, say, an array of pixels. But instead of pixels each element of the array is some kind of record structure (for example two floats, a boolean, and a 20 character string field). Just as a 64-bit float is an 8 byte record in an array, a row in such a table can be thought of as a 37 byte (in the case of the previous example) record in a 1-D array of rows. So in principle everything that was explained in the answer to the question “[What performance differences are there between astropy.io.fits and fitsio?](#)” applies just as well to tables as it does to any other array.

However, FITS tables have many additional complexities that sometimes preclude streaming the data directly from disk, and instead require transformation from the on-disk FITS format to a format more immediately useful to the user. A common example is how FITS represents boolean values in binary tables. Another, significantly more complicated example, is variable length arrays.

As explained in “[Why is reading rows out of a FITS table so slow?](#)”, `astropy.io.fits/PyFITS` does not currently handle some of these cases as efficiently as it could, in particular in cases where a user only wishes to read a few rows out of a table. Fitsio, on the other hand, has a better interface for copying one row at a time out of a table and performing the necessary transformations on that row *only*, rather than on the entire column or columns that the row is taken from. As such, for many cases fitsio gets much better performance and should be preferred for many performance-critical table operations.

Fitsio also exposes a microlanguage (implemented in CFITSIO) for making efficient SQL-like queries of tables (single tables only though—no joins or anything like that). This format, described in the [CFITSIO documentation](#) can in some cases perform more efficient selections of rows than might be possible with Numpy alone, which requires creating an intermediate mask array in order to perform row selection.

14.4.2 Header Interface Transition Guide

Note: This guide was originally included with the release of PyFITS 3.1, and still references PyFITS in many places, though the examples have been updated for `astropy.io.fits`. It is still useful here for informational purposes, though Astropy has always used the PyFITS 3.1 Header interface.

PyFITS v3.1 included an almost complete rewrite of the `Header` interface. Although the new interface is largely compatible with the old interace (whether due to similarities in the design, or backwards-compatibility support), there are enough differences that a full explanation of the new interface is merited.

The Trac ticket discussing the initial motivation and changes to be made to the `Header` class is [#64](#). It may be worth reading for some of the background to this work, though this document contains a more complete description of the “final” product (which will continue to evolve).

Background

Prior to 3.1, PyFITS users interacted with FITS headers by way of three different classes: `Card`, `CardList`, and `Header`.

The `Card` class represents a single header card with a keyword, value, and comment. It also contains all the machinery for parsing FITS header cards, given the 80 character string, or “card image” read from the header.

The `CardList` class is actually a subclass of Python’s `list` built-in. It was meant to represent the actual list of cards that make up a header. That is, it represents an ordered list of cards in the physical order that they appear in the header. It supports the usual list methods for inserting and appending new cards into the list. It also supports `dict`-like keyword access, where `cardlist['KEYWORD']` would return the first card in the list with the given keyword.

A lot of the functionality for manipulating headers was actually buried in the `CardList` class. The `Header` class was more of a wrapper around `CardList` that added a little bit of abstraction. It also implemented a partial `dict`-like interface, though for `Header`s a keyword lookup returned the header value associated with that keyword, and not the `Card` object. Though almost every method on the `Header` class was just performing some operations on the underlying `CardList`.

The problem is that there were certain things one could *only* do by directly accessing the `CardList`, such as look up the comments on a card, or access cards that have duplicate keywords, such as `HISTORY`. Another long-standing misfeature that slicing a `Header` object actually returned a `CardList` object, rather than a new `Header`. For all but the most simple use cases, working with `CardList` objects was largely unavoidable.

But it was realized that `CardList` is really an implementation detail not representing any element of a FITS file distinct from the header itself. Users familiar with the FITS format know what a header is, but it’s not clear how a “card list” is distinct from that, or why operations go through the `Header` object, while some have to be performed through the `CardList`.

So the primary goal of this redesign was eliminate the `CardList` class altogether, and make it possible for users to perform all header manipulations directly through `Header` objects. It also tries to present headers as similar as possible to more a more familiar data structure—an ordered mapping (or `OrderedDict` in Python) for ease of use by new users less familiar with the FITS format. Though there are still many added complexities for dealing with the idiosyncracies of the FITS format.

Deprecation Warnings

A few old methods on the `Header` class have been marked as deprecated, either because they have been renamed to a more [PEP 8](#)-compliant name, or because have become redundant due to new features. To check if your code is using any deprecated methods or features, run your code with `python -Wd`. This will output any deprecation warnings to the console.

Two of the most common deprecation warnings related to `Header`s are for:

- `:meth:Header.has_key`—this has actually been deprecated since PyFITS 3.0, just as Python’s `dict.has_key` is deprecated. For checking a key’s presence in a mapping object like `dict` or `Header`, use the `key in d` syntax. This has long been the preference in Python.
- `:meth:Header.ascardlist` and `Header.ascard`—these were used to access the `CardList` object underlying a header. They should still work, and return a skeleton `CardList` implementation that should support most of the old `CardList` functionality. But try removing as much of this as possible. If direct access to the `Card` objects making up a header is necessary, use `Header.cards`, which returns an iterator over the cards. More on that below.

New Header Design

The new `Header` class is designed to work as a drop-in replacement for a `dict` via *duck typing*. That is, although it is not a subclass of `dict`, it implements all the same methods and interfaces. In particular, it is similar to an `OrderedDict` in that the order of insertions is preserved. However, `Header` also supports many additional features and behaviors specific to the FITS format. It should also be noted that while the old `Header` implementation also had a dict-like interface, it did not implement the *entire* dict interface as the new `Header` does.

Although the new `Header` is used like a dict/mapping in most cases, it also supports a `list` interface. The list-like interface is a bit idiosyncratic in that in some contexts the `Header` acts like a list of values, in some like a list of keywords, and in a few contexts like a list of `Card` objects. This may be the most difficult aspect of the new design, but there is logic to it.

As with the old `Header` implementation, integer index access is supported: `header[0]` returns the value of the first keyword. However, the `Header.index()` method treats the header as though it’s a list of keywords, and returns the index of a given keyword. For example:

```
>>> header.index('BITPIX')
2
```

`Header.count()` is similar to `list.count`, and also takes a keyword as its argument:

```
>>> header.count('HISTORY')
20
```

A good rule of thumb is that any item access using square brackets `[]` returns *value* in the header, whether using keyword or index lookup. Methods like `index()` and `count()` that deal with the order and quantity of items in the `Header` generally work on keywords. Finally, methods like `insert()` and `append()` that add new items to the header work on cards.

Aside from the list-like methods, the new `Header` class works very similarly to the old implementation for most basic use cases and should not present too many surprises. There are differences, however:

- As before, the `Header()` initializer can take a list of `Card` objects with which to fill the header. However, now any iterable may be used. It is also important to note that *any* `Header` method that accepts `Card` objects can also accept 2-tuples or 3-tuples in place of `Cards`. That is, either a `(keyword, value, comment)` tuple or a `(keyword, value)` tuple (comment is assumed blank) may be used anywhere in place of a `Card` object. This is even preferred, as it simply involves less typing. For example:

```
>>> from astropy.io import fits
>>> header = fits.Header([('A', 1), ('B', 2), ('C', 3, 'A comment')])
>>> header
A      =          1
B      =          2
C      =          3 / A comment
```

- As demonstrated in the previous example, the `repr()` for a `Header`, that is the text that is displayed when entering a `Header` object in the Python console as an expression, shows the header as it would appear in a FITS

file. This inserts newlines after each card so that it is easily readable regardless of terminal width. It is *not* necessary to use `print header` to view this. Simply entering `header` displays the header contents as it would appear in the file (sans the END card).

- `len(header)` is now supported (previously it was necessary to do `len(header.ascard)`). This returns the total number of cards in the header, including blank cards, but excluding the END card.
- FITS supports having duplicate keywords, although they are generally in error except for commentary keywords like COMMENT and HISTORY. PyFITS now supports reading, updating, and deleting duplicate keywords: Instead of using the keyword by itself, use a `(keyword, index)` tuple. For example `('HISTORY', 0)` represents the first HISTORY card, `('HISTORY', 1)` represents the second HISTORY card, and so on. In fact, when a keyword is used by itself, it's really just shorthand for `(keyword, 0)`. It is now possible to delete an accidental duplicate like so:

```
>>> del header[('NAXIS', 1)]
```

This will remove an accidental duplicate NAXIS card from the header.

- Even if there are duplicate keywords, keyword lookups like `header['NAXIS']` will always return the value associated with the first copy of that keyword, with one exception: Commentary keywords like COMMENT and HISTORY are expected to have duplicates. So `header['HISTORY']`, for example, returns the whole sequence of HISTORY values in the correct order. This list of values can be sliced arbitrarily. For example, to view the last 3 history entries in a header:

```
>>> hdulist[0].header['HISTORY'][-3:]
reference table oref$laf13367o_pct.fits
reference table oref$laf13369o_apt.fits
Heliocentric correction = 16.225 km/s
```

- Subscript assignment can now be used to add new keywords to the header. Just as with a normal `dict`, `header['NAXIS'] = 1` will either update the NAXIS keyword if it already exists, or add a new NAXIS keyword with a value of 1 if it does not exist. In the old interface this would return a `KeyError` if NAXIS did not exist, and the only way to add a new keyword was through the `update()` method.

By default, new keywords added in this manner are added to the end of the header, with a few FITS-specific exceptions:

- If the header contains extra blank cards at the end, new keywords are added before the blanks.
- If the header ends with a list of commentary cards—for example a sequence of HISTORY cards—those are kept at the end, and new keywords are inserted before the commentary cards.
- If the keyword is a commentary keyword like COMMENT or HISTORY (or an empty string for blank keywords), a *new* commentary keyword is always added, and appended to the last commentary keyword of the same type. For example, HISTORY keywords are always placed after the last history keyword:

```
>>> header = fits.Header()
>>> header['COMMENT'] = 'Comment 1'
>>> header['HISTORY'] = 'History 1'
>>> header['COMMENT'] = 'Comment 2'
>>> header['HISTORY'] = 'History 2'
>>> header
COMMENT Comment 1
COMMENT Comment 2
HISTORY History 1
HISTORY History 2
```

These behaviors represent a sensible default behavior for keyword assignment, and represents the same behavior as `update()` in the old Header implementation. The default behaviors may still be bypassed through the use of other assignment methods like `Header.set()` and `Header.append()` described later.

- It is now also possible to assign a value and a comment to a keyword simultaneously using a tuple:

```
>>> header['NAXIS'] = (2, 'Number of axis')
```

This will update the value and comment of an existing keyword, or add a new keyword with the given value and comment.

- There is a new `Header.comments` attribute which lists all the comments associated with keywords in the header (not to be confused with COMMENT cards). This allows viewing and updating the comments on specific cards:

```
>>> header.comments['NAXIS']
Number of axis
>>> header.comments['NAXIS'] = 'Number of axes'
>>> header.comments['NAXIS']
Number of axes
```

- When deleting a keyword from a header, don't assume that the keyword already exists. In the old Header implementation this would just silently do nothing. For backwards-compatibility it is still okay to delete a non-existent keyword, but a warning will be raised. In the future this *will* be changed so that trying to delete a non-existent keyword raises a `KeyError`. This is for consistency with the behavior of Python dicts. So unless you know for certain that a keyword exists before deleting it, it's best to do something like:

```
>>> try:
...     del header['BITPIX']
... except KeyError:
...     pass
```

Or if you prefer to look before you leap:

```
>>> if 'BITPIX' in header:
...     del header['BITPIX']
```

- `del header` now supports slices. For example, to delete the last three keywords from a header:

```
>>> del header[-3:]
```

- Two headers can now be compared for equality—previously no two Header objects were the same. Now they compare as equal if they contain the exact same content. That is, this requires strict equality.
- Two headers can now be added with the '+' operator, which returns a copy of the left header extended by the right header with `extend()`. Assignment addition is also possible.
- The `Header.update()` method used commonly with the old Header API has been renamed to `Header.set()`. The primary reason for this change is very simple: Header implements the `dict` interface, which already has a method called `update()`, but that behaves differently from the old `Header.update()`.

The details of the new `update()` can be read in the API docs, but it is very similar to `dict.update`. It also supports backwards compatibility with the old `update()` by analysis of the arguments passed to it, so existing code will not break immediately. However, this *will* cause a deprecation warning to be output if they're enabled. It is best, for starters, to replace all `update()` calls with `set()`. Recall, also, that direct assignment is now possible for adding new keywords to a header. So by and large the only reason to prefer using `Header.set()` is its capability of inserting or moving a keyword to a specific location using the `before` or `after` arguments.

- Slicing a Header with a slice index returns a new Header containing only those cards contained in the slice. As mentioned earlier, it used to be that slicing a Header returned a card list—something of a misfeature. In general, objects that support slicing ought to return an object of the same type when you slice them.

Likewise, wildcard keywords used to return a `CardList` object. Now they return a new Header—similarly to a slice. For example:

```
>>> header['NAXIS*']
```

returns a new header containing only the NAXIS and NAXISn cards from the original header.

Transition Tips

The above may seem like a lot, but the majority of existing code using PyFITS to manipulate headers should not need to be updated, at least not immediately. The most common operations still work the same.

As mentioned above, it would be helpful to run your code with `python -Wd` to enable deprecation warnings—that should be a good idea of where to look to update your code.

If your code needs to be able to support older versions of PyFITS simultaneously with PyFITS 3.1, things are slightly trickier, but not by much—the deprecated interfaces will not be removed for several more versions because of this.

- The first change worth making, which is supported by any PyFITS version in the last several years, is remove any use of `:meth:Header.has_key` and replace it with `keyword in header` syntax. It’s worth making this change for any dict as well, since `dict.has_key` is deprecated. Running the following regular expression over your code may help with most (but not all) cases:

```
s/([\^ ]+)\.has_key\([\^ ]+\)\//\2 in \1/
```

- If possible, replace any calls to `Header.update()` with `Header.set()` (though don’t bother with this if you need to support older PyFITS versions). Also, if you have any calls to `Header.update()` that can be replaced with simple subscript assignments (eg. `header['NAXIS'] = (2, 'Number of axes')`) do that too, if possible.
- Find any code that uses `header.ascard` or `header.ascardlist()`. First ascertain whether that code really needs to work directly on `Card` objects. If that is definitely the case, go ahead and replace those with `header.cards`—that should work without too much fuss. If you do need to support older versions, you may keep using `header.ascard` for now.
- In the off chance that you have any code that slices a header, it’s best to take the result of that and create a new `Header` object from it. For example:

```
>>> new_header = fits.Header(old_header[2:])
```

This avoids the problem that in PyFITS ≤ 3.0 slicing a `Header` returns a `CardList` by using the result to initialize a new `Header` object. This will work in both cases (in PyFITS 3.1, initializing a `Header` with an existing `Header` just copies it, a la `list`).

- As mentioned earlier, locate any code that deletes keywords with `del`, and make sure they either look before they leap (`if keyword in header:`) or ask forgiveness (`try/except KeyError:`).

Other Gotchas

- As mentioned above it is not necessary to enter `print header` to display a header in an interactive Python prompt. Simply entering `>>> header` by itself is sufficient. Using `print` usually will *not* display the header readably, because it does not include line-breaks between the header cards. The reason is that Python has two types of string representations: One is returned when one calls `str(header)` which happens automatically when you `print` a variable. In the case of the `Header` class this actually returns the string value of the header as it is written literally in the FITS file, which includes no line breaks.

The other type of string representation happens when one calls `repr(header)`. The `repr` of an object is just meant to be a useful string “representation” of the object; in this case the contents of the header but with linebreaks between the cards and with the END card and padding trailing padding stripped off. This happens automatically when one enters a variable at the Python prompt by itself without a `print` call.

- The current version of the FITS Standard (3.0) states in section 4.2.1 that trailing spaces in string values in headers are not significant and should be ignored. PyFITS < 3.1 *did* treat trailing spaces as significant. For example if a header contained:

```
KEYWORD1= 'Value '
```

then `header['KEYWORD1']` would return the string `'Value '` exactly, with the trailing spaces intact. The new Header interface fixes this by automatically stripping trailing spaces, so that `header['KEYWORD1']` would return just `'Value'`.

There is, however, one convention used by the IRAF ccdmosiac task for representing its [TNX World Coordinate System](#) and [ZPX World Coordinate System](#) non-standard WCS' that uses a series of keywords in the form `WATj_nnn` which store a text description of coefficients for a non-linear distortion projection. It uses its own microformat for listing the coefficients as a string, but the string is long, and thus broken up into several of these `WATj_nnn` keywords. Correct recombination of these keywords requires treating all whitespace literally. This convention either overlooked or predated the prescribed treatment of whitespace in the FITS standard.

To get around this issue a global variable `fits.STRIP_HEADER_WHITESPACE` was introduced. Temporarily setting `fits.STRIP_HEADER_WHITESPACE.set(False)` before reading keywords affected by this issue will return their values with all trailing whitespace intact.

A future version of PyFITS may be able to detect use of conventions like this contextually and behave according to the convention, but in most cases the default behavior of PyFITS is to behave according to the FITS Standard.

14.4.3 `astropy.io.fits` History

Prior to its inclusion in Astropy, the `astropy.io.fits` package was a stand-alone package called `PyFITS`. Though for the time being active development is continuing on PyFITS, that development is also being merged into Astropy. This page documents the release history of PyFITS prior to its merge into Astropy.

PyFITS Changelog

- 3.3.0 (unreleased)
 - New Features
 - API Changes
 - Other Changes and Additions
 - Bug Fixes
- 3.2.4 (unreleased)
- 3.1.7 (unreleased)
- 3.2.3 (2014-05-14)
- 3.1.6 (2014-05-14)
- 3.2.2 (2014-03-25)
- 3.1.5 (2014-03-25)
- 3.2.1 (2014-03-04)
- 3.1.4 (2014-03-04)
- 3.0.13 (2014-03-04)
- 3.2 (2013-11-26)
 - Highlights
 - API Changes
 - Other Changes and Additions
 - Bug Fixes
- 3.1.3 (2013-11-26)
- 3.0.12 (2013-11-26)
- 3.1.3 (unreleased)
- 3.0.12 (unreleased)
- 3.1.2 (2013-04-22)
- 3.0.11 (2013-04-17)
- 3.1.1 (2013-01-02)
 - Bug Fixes
- 3.0.10 (2013-01-02)
- 3.1 (2012-08-08)
 - Highlights
 - API Changes
 - New Features
 - Changes in Behavior
 - Bug Fixes
- 3.0.9 (2012-08-06)
 - Bug Fixes
- 3.0.8 (2012-06-04)
 - Changes in Behavior
 - Bug Fixes
- 3.0.7 (2012-04-10)
 - Changes in Behavior
 - Bug Fixes
- 3.0.6 (2012-02-29)
 - Highlights
 - Bug Fixes
- 3.0.5 (2012-01-30)
- 3.0.4 (2011-11-22)
- 3.0.3 (2011-10-05)
- 3.0.2 (2011-09-23)
- 3.0.1 (2011-09-12)
- 3.0.0 (2011-08-23)
- 2.4.0 (2011-01-10)
- 2.3.1 (2010-06-03)
- 2.3 (2010-05-11)

14.4. Other Information

- 2.2.1 (2009-10-06)
- 2.2 (2009-09-23)
- 2.1.1 (2009-04-22)
- 2.1 (2009-04-14)

3.3.0 (unreleased)

New Features

- Added new verification options `fix+ignore`, `fix+warn`, `fix+exception`, `silentfix+ignore`, `silentfix+warn`, and `silentfix+exception` which give more control over how to report fixable errors as opposed to unfixable errors. See the “Verification” section in the PyFITS documentation for more details.

API Changes

- The `pyfits.new_table` function is now fully deprecated (though will not be removed for a long time, considering how widely it is used).

Instead please use the more explicit `pyfits.BinTableHDU.from_columns` to create a new binary table HDU, and the similar `pyfits.TableHDU.from_columns` to create a new ASCII table. These otherwise accept the same arguments as `pyfits.new_table` which is now just a wrapper for these.

- The `.fromstring` classmethod of each HDU type has been simplified such that, true to its namesake, it only initializes an HDU from a string containing its header *and* data. (spacetelescope/PyFITS#64)
- Fixed an issue where header wildcard matching (for example `header['DATE*']`) can be used to match *any* characters that might appear in a keyword. Previously this only matched keywords containing characters in the set `[0-9A-Za-z_]`. Now this can also match a hyphen `-` and any other characters, as some conventions like `HIERARCH` and record-valued keyword cards allow a wider range of valid characters than standard FITS keywords.
- This will be the *last* release to support the following APIs that have been marked deprecated since PyFITS v3.1:
 - The `CardList` class, which was part of the old header implementation.
 - The `Card.key` attribute. Use `Card.keyword` instead.
 - The `Card.cardimage` and `Card.ascardimage` attributes. Use simply `Card.image` or `str(card)` instead.
 - The `create_card` factory function. Simply use the normal `Card` constructor instead.
 - The `create_card_from_string` factory function. Use `Card.fromstring` instead.
 - The `upper_key` function. Use `Card.normalize_keyword` method instead (this is not unlikely to be used outside of PyFITS itself, but it was technically public API).
 - The usage of `Header.update` with `Header.update(keyword, value, comment)` arguments. `Header.update` should only be used analogously to `dict.update`. Use `Header.set` instead.
 - The `Header.ascard` attribute. Use `Header.cards` instead for a list of all the `Card` objects in the header.
 - The `Header.rename_key` method. Use `Header.rename_keyword` instead.
 - The `Header.get_history` method. Use `header['HISTORY']` instead (normal keyword lookup).
 - The `Header.get_comment` method. Use `header['COMMENT']` instead.
 - The `Header.toTxtFile` method. Use `header.totextfile` instead.
 - The `Header.fromTxtFile` method. Use `Header.fromtextfile` instead.
 - The `pyfits.tdump` and `tcreate` functions. Use `pyfits.tabledump` and `pyfits.tableload` respectively.

- The `BinTableHDU.tdump` and `tcreate` methods. Use `BinTableHDU.dump` and `BinTableHDU.load` respectively.
- The `txtfile` argument to the `Header` constructor. Use `Header.fromfile` instead.
- The `startColumn` and `endColumn` arguments to the `FITS_record` constructor. These are unlikely to be used by any user code.

These deprecated interfaces will be removed from the development version of PyFITS following the v3.3 release (they will still be available in any v3.3.x bugfix releases, however).

Other Changes and Additions

- PyFITS has switched to a unified code base which supports Python 2.5 through 3.4 simultaneously without translation. This *shouldn't* have any significant performance impacts, but please report if anything seems noticeably slower. As a reminder, support for Python 2.5 will be ended after PyFITS 3.3.x.
- Warnings for deprecated APIs in PyFITS are now always displayed by default. This is in line with a similar change made recently to Astropy: <https://github.com/astropy/astropy/pull/1871> To disable PyFITS deprecation warnings in scripts one may call `pyfits.ignore_deprecation_warnings()` after importing PyFITS.
- `Card` objects have a new `is_blank` attribute which returns `True` if the card represents a blank card (no keyword, value, or comment) and `False` otherwise.

Bug Fixes

- Fixed a regression where it was not possible to save an empty “compressed” image to a file (in this case there is nothing to compress, hence the quotes, but trying to do so caused a crash). (spacetelescope/PyFITS#69)
- Fixed a regression that may have been introduced in v3.2.1 with writing compressed image HDUs, particularly compressed images using a non-empty `GZIP_COMPRESSED_DATA` column. (spacetelescope/#71)

3.2.4 (unreleased)

- Fixed a regression where multiple consecutive calls of the `writeto` method on the same HDU but to different files could lead to corrupt data or crashes on the subsequent calls after the first. (spacetelescope/PyFITS#40)

3.1.7 (unreleased)

- Nothing changed yet.

3.2.3 (2014-05-14)

- Nominal support for Python 3.4.
- Fixed a bug with using the `tabledump` and `tableload` functions with tables containing array columns (columns in which each element is an array instead of a single scalar value). (spacetelescope/PyFITS#22)
- Fixed an issue where PyFITS allowed newline characters in header values and comments. (spacetelescope/PyFITS#51)
- Fixed pickling of `FITS_rec` (table data) objects. (spacetelescope/PyFITS#53)
- Improved behavior when writing large compressed images on OSX by removing an unnecessary check for platform architecture. (spacetelescope/PyFITS#57)

- Allow reading FITS files from file-like objects that do not have a `.closed` attribute (and as such may not even have an “open” vs. “closed” concept). (spacetelescope/PyFITS#56)
- Fixed duplicate insertion of commentary keywords on compressed image headers. (spacetelescope/PyFITS#58)
- Fixed minor issue with comparison of header commentary card values. (spacetelescope/PyFITS#59)

3.1.6 (2014-05-14)

- Nominal support for Python 3.4.
- Fixed a bug with using the `tabledump` and `tableload` functions with tables containing array columns (columns in which each element is an array instead of a single scalar value). (Backported from 3.2.3)
- Fixed an issue where PyFITS allowed newline characters in header values and comments. (Backported from 3.2.3)
- Fixed pickling of `FITS_rec` (table data) objects. (Backported from 3.2.3)
- Improved behavior when writing large compressed images on OSX by removing an unnecessary check for platform architecture. (Backported from 3.2.3)
- Allow reading FITS files from file-like objects that do not have a `.closed` attribute (and as such may not even have an “open” vs. “closed” concept). (Backported from 3.2.3)
- Fixed minor issue with comparison of header commentary card values. (Backported from 3.2.3)

3.2.2 (2014-03-25)

- Fixed a regression on deletion of record-valued keyword cards using the Header wildcard syntax. This was intended to be fixed before the v3.2.1 release.

3.1.5 (2014-03-25)

- Fixed a regression on deletion of record-valued keyword cards using the Header wildcard syntax. This was intended to be fixed before the v3.1.4 release.

3.2.1 (2014-03-04)

- Nominal support for the upcoming Python 3.4.
- Added missing features from the `Header.insert()` method that were intended for inclusion in the original 3.1 release: In addition to accepting an integer index as the first argument, it also supports supplying a keyword name as the first argument for insertion relative to a specific keyword. It also now supports an optional `after` argument. If `after=True` the the insertion is made below the insertion point instead of above it. (spacetelescope/PyFITS#12)
- Fixed support for broadcasting of values assigned to table columns. (spacetelescope/PyFITS#48)
- A grab bag of minor performance improvements in headers. (spacetelescope/PyFITS#46)
- Fix an unrelated error that occurred when instantiating a `ColDefs` object with invalid input.
- Fixed an issue where opening an image containing pseudo-unsigned integers and immediately writing it to a new file using the `writeto` method would drop the scale factors that identified the data as unsigned.
- Fixed a bug where writing a file with `checksum=True` did not add the checksum on new files. (spacetelescope/PyFITS#8)

- Fixed an issue where validating an HDU’s checksums removed the checksum from that HDU’s header entirely (even if it was valid.)
- Fixed checksums on compressed images, so that the `ZCHECKSUM` and `ZDATASUM` contain a checksum of the original image HDU, while `CHECKSUM` and `DATASUM` contain checksums of the compressed image HDU. This feature was supposed to be supported in 3.2, but the support was buggy.
- Fixed an issue where the size of the heap was sometimes not computed properly when writing an existing table containing variable-length array columns to a new FITS file. This could result in corruption in the new FITS file. (spacetelescope/PyFITS#47)
- Fixed issue with updates to the header of `CompImageHDU` objects not being preserved on save. (spacetelescope/PyFITS#23)
- Fixed a bug where a boolean value of `True` in a header could not be replaced with the integer 1, and likewise for `False` and 0 and vice versa.
- Fixed an issue similar to the above one but for numeric values—now replacing a header value with an equivalent numeric value will up/downcast that value. For example replacing ‘0’ with ‘0.0’ will write ‘0.0’ to the header so that it is returned as a floating point value. Likewise a float can be downcast to an integer. (spacetelescope/PyFITS#49)
- A handful of Python 3 compatibility fixes, especially for compatibility with the upcoming Python 3.4.
- Fixed unrelated crash when a header contains an invalid END card (for example “END = ”). This resulted in a cryptic traceback. Now headers like this will detect “clearly intended” END cards and produce a warning about their invalidity and fix them. (#217)
- Allowed a sequence of `Column` objects to be passed in as the main argument to `FITS_rec.from_columns` as the documentation suggests should be possible.
- Fixed a display formatting issue with `fitsdiff` where sometimes it did not show the difference between two floating point numbers if they were the same up to some low number of digits. (spacetelescope/PyFITS#21)
- Fixed an issue where Python 2 sometimes allowed non-ASCII strings to be assigned as header values if they were assigned as old-style `str` objects and not `unicode` objects. (spacetelescope/PyFITS#37)

3.1.4 (2014-03-04)

- Added missing features from the `Header.insert()` method that were intended for inclusion in the original 3.1 release: In addition to accepting an integer index as the first argument, it also supports supplying a keyword name as the first argument for insertion relative to a specific keyword. It also now supports an optional `after` argument. If `after=True` the the insertion is made below the insertion point instead of above it. (Backported from 3.2.1)
- A grab bag of minor performance improvements in headers. (Backported from 3.2.1)
- Fixed an issue where opening an image containing pseudo-unsigned integers and immediately writing it to a new file using the `writeto` method would drop the scale factors that identified the data as unsigned. (Backported from 3.2.1)
- Fixed a bug where writing a file with `checksum=True` did not add the checksum on new files. (Backported from 3.2.1)
- Fixed an issue where validating an HDU’s checksums removed the checksum from that HDU’s header entirely (even if it was valid.) (Backported from 3.2.1)
- Fixed an issue where the size of the heap was sometimes not computed properly when writing an existing table containing variable-length array columns to a new FITS file. This could result in corruption in the new FITS file. (Backported from 3.2.1)

- Fixed a bug where a boolean value of `True` in a header could not be replaced with the integer 1, and likewise for `False` and 0 and vice versa. (Backported from 3.2.1)
- Fixed an issue similar to the above one but for numeric values—now replacing a header value with an equivalent numeric value will up/downcast that value. For example replacing `'0'` with `'0.0'` will write `'0.0'` to the header so that it is returned as a floating point value. Likewise a float can be downcast to an integer. (Backported from 3.2.1)
- Fixed unrelated crash when a header contains an invalid END card (for example `"END = "`). This resulted in a cryptic traceback. Now headers like this will detect “clearly intended” END cards and produce a warning about their invalidity and fix them. (Backported from 3.2.1)
- Fixed a display formatting issue with `fitsdiff` where sometimes it did not show the difference between two floating point numbers if they were the same up to some low number of digits. (Backported from 3.2.1)
- Fixed an issue where Python 2 sometimes allowed non-ASCII strings to be assigned as header values if they were assigned as old-style `str` objects and not `unicode` objects. (Backported from 3.2.1)

3.0.13 (2014-03-04)

- Fixed a bug where writing a file with `checksum=True` did not add the checksum on new files. (Backported from 3.2.1)
- Fixed an issue where validating an HDU’s checksums removed the checksum from that HDU’s header entirely (even if it was valid.) (Backported from 3.2.1)

3.2 (2013-11-26)

Highlights

- Rewrote CFITSIO-based backend for handling tile compression of FITS files. It now uses a standard CFITSIO instead of heavily modified pieces of CFITSIO as before. PyFITS ships with its own copy of CFITSIO v3.35 which supports the latest version of the Tiled Image Convention (v2.3), but system packagers may choose instead to strip this out in favor of a system-installed version of CFITSIO. Earlier versions may work, but nothing earlier than 3.28 has been tested yet. (#169)
- Added support for reading and writing tables using the Q format for columns. The Q format is identical to the P format (variable-length arrays) except that it uses 64-bit integers for the data descriptors, allowing more than 4 GB of variable-length array data in a single table. (#160)
- Added initial support for table columns containing pseudo-unsigned integers. This is currently enabled by using the `uint=True` option when opening files; any table columns with the correct BZERO value will be interpreted and returned as arrays of unsigned integers.
- Some refactoring of the table and `FITS_rec` modules in order to better separate the details of the FITS binary and ASCII table data structures from the HDU data structures that encapsulate them. Most of these changes should not be apparent to users (but see API Changes below).

API Changes

- Assigning to values in `ColDefs.names`, `ColDefs.formats`, `ColDefs.nulls` and other attributes of `ColDefs` instances that return lists of column properties is no longer supported. Assigning to those lists will no longer update the corresponding columns. Instead, please just modify the `Column` instances directly (`Column.name`, `Column.null`, etc.)

- The `pyfits.new_table` function is marked “pending deprecation”. This does not mean it will be removed outright or that its functionality has changed. It will likely be replaced in the future for a function with similar, if not subtly different functionality. A better, if not slightly more verbose approach is to use `pyfits.FITS_rec.from_columns` to create a new `FITS_rec` table—this has the same interface as `pyfits.new_table`. The difference is that it returns a plain `FITS_rec` array, and not an HDU instance. This `FITS_rec` object can then be used as the data argument in the constructors for `BinTableHDU` (for binary tables) or `TableHDU` (for ASCII tables). This is analogous to creating an `ImageHDU` by passing in an image array. `pyfits.FITS_rec.from_columns` is just a simpler way of creating a FITS-compatible recarray from a FITS column specification.
- The `updateHeader`, `updateHeaderData`, and `updateCompressedData` methods of the `CompDataHDU` class are pending deprecation and moved to internal methods. The operation of these methods depended too much on internal state to be used safely by users; instead they are invoked automatically in the appropriate places when reading/writing compressed image HDUs.
- The `CompDataHDU.compData` attribute is pending deprecation in favor of the clearer and more PEP-8 compatible `CompDataHDU.compressed_data`.
- The constructor for `CompDataHDU` has been changed to accept new keyword arguments. The new keyword arguments are essentially the same, but are in underscore_separated format rather than camelCase format. The old arguments are still pending deprecation.
- The internal attributes of HDU classes `_hdrLoc`, `_datLoc`, and `_datSpan` have been replaced with `_header_offset`, `_data_offset`, and `_data_size` respectively. The old attribute names are still pending deprecation. This should only be of interest to advanced users who have created their own HDU subclasses.
- The following previously deprecated functions and methods have been removed entirely: `createCard`, `createCardFromString`, `upperKey`, `ColDefs.data`, `setExtensionNameCaseSensitive`, `_File.getFile`, `_TableBaseHDU.get_coldefs`, `Header.has_key`, `Header.ascardlist`.
If you run your code with a previous version of PyFITS (≥ 3.0 , < 3.2) with the `python -Wd` argument, warnings for all deprecated interfaces still in use will be displayed.
- Interfaces that were pending deprecation are now fully deprecated. These include: `create_card`, `create_card_from_string`, `upper_key`, `Header.get_history`, and `Header.get_comment`.
- The `.name` attribute on HDUs is now directly tied to the HDU’s header, so that if `.header['EXTNAME']` changes so does `.name` and vice-versa.
- The `pyfits.file.PYTHON_MODES` constant dict was renamed to `pyfits.file.PYFITS_MODES` which better reflects its purpose. This is rarely used by client code, however. Support for the old name will be removed by PyFITS 3.4.

Other Changes and Additions

- The new compression code also adds support for the `ZQUANTIZ` and `ZDITHER0` keywords added in more recent versions of this FITS Tile Compression spec. This includes support for lossless compression with GZIP. (#198) By default no dithering is used, but the `SUBTRACTIVE_DITHER_1` and `SUBTRACTIVE_DITHER_2` methods can be enabled by passing the correct constants to the `quantize_method` argument to the `CompImageHDU` constructor. A seed can be manually specified, or automatically generated using either the system clock or checksum-based methods via the `dither_seed` argument. See the documentation for `CompImageHDU` for more details. (#198) ([spacetelescope/PYFITS#32](#))
- Images compressed with the Tile Compression standard can now be larger than 4 GB through support of the Q format. (#159)

- All HDUs now have a `.ver.level` attribute that returns the value of the `EXTVAL` and `EXTLEVEL` keywords from that HDU's header, if they exist. This was added for consistency with the `.name` attribute which returns the `EXTNAME` value from the header.
- Then `Column` and `ColDefs` classes have new `.dtype` attributes which give the Numpy dtype for the column data in the first case, and the full Numpy compound dtype for each table row in the latter case.
- There was an issue where new tables created defaulted the values in all string columns to '0.0'. Now string columns are filled with empty strings by default—this seems a less surprising default, but it may cause differences with tables created with older versions of PyFITS.
- Improved round-tripping and preservation of manually assigned column attributes (`TNULLn`, `TSCALn`, etc.) in table HDU headers. ([astropy/astropy#996](#))

Bug Fixes

- Binary tables containing compressed images may, optionally, contain other columns unrelated to the tile compression convention. Although this is an uncommon use case, it is permitted by the standard. ([#159](#))
- Reworked some of the file I/O routines to allow simpler, more consistent mapping between OS-level file modes ('rb', 'wb', 'ab', etc.) and the more "PyFITS-specific" modes used by PyFITS like "readonly" and "update". That is, if reading a FITS file from an open file object, it doesn't matter as much what "mode" it was opened in so long as it has the right capabilities (read/write/etc.) Also works around bugs in the Python io module in 2.6+ with regard to file modes. ([spacetelescope/PyFITS#33](#))
- Fixed an obscure issue that can occur on systems that don't have flush to memory-mapped files implemented (namely GNU Hurd). ([astropy/astropy#968](#))

3.1.3 (2013-11-26)

- Disallowed assigning NaN and Inf floating point values as header values, since the FITS standard does not define a way to represent them in. Because this is undefined, the previous behavior did not make sense and produced invalid FITS files. ([spacetelescope/PyFITS#11](#))
- Added a workaround for a bug in 64-bit OSX that could cause truncation when writing files greater than 2³² bytes in size. ([spacetelescope/PyFITS#28](#))
- Fixed a long-standing issue where writing binary tables did not correctly write the `TFORMn` keywords for variable-length array columns (they omitted the max array length parameter of the format). This was thought fixed in v3.1.2, but it was only fixed there for compressed image HDUs and not for binary tables in general.
- Fixed an obscure issue that can occur on systems that don't have flush to memory-mapped files implemented (namely GNU Hurd). (Backported from 3.2)

3.0.12 (2013-11-26)

- Disallowed assigning NaN and Inf floating point values as header values, since the FITS standard does not define a way to represent them in. Because this is undefined, the previous behavior did not make sense and produced invalid FITS files. (Backported from 3.1.3)
- Added a workaround for a bug in 64-bit OSX that could cause truncation when writing files greater than 2³² bytes in size. (Backported from 3.1.3)
- Fixed a long-standing issue where writing binary tables did not correctly write the `TFORMn` keywords for variable-length array columns (they omitted the max array length parameter of the format). This was thought fixed in v3.1.2, but it was only fixed there for compressed image HDUs and not for binary tables in general. (Backported from 3.1.3)

- Fixed an obscure issue that can occur on systems that don't have flush to memory-mapped files implemented (namely GNU Hurd). (Backported from 3.2)

3.1.3 (unreleased)

- Disallowed assigning NaN and Inf floating point values as header values, since the FITS standard does not define a way to represent them in. Because this is undefined, the previous behavior did not make sense and produced invalid FITS files. (spacetelescope/PyFITS#11)

3.0.12 (unreleased)

- Disallowed assigning NaN and Inf floating point values as header values, since the FITS standard does not define a way to represent them in. Because this is undefined, the previous behavior did not make sense and produced invalid FITS files. (Backported from 3.1.3)
- Added a workaround for a bug in 64-bit OSX that could cause truncation when writing files greater than 2^{32} bytes in size. (Backported from 3.1.3)

3.1.2 (2013-04-22)

- When an error occurs opening a file in fitsdiff the exception message will now at least mention which file had the error. (#168)
- Fixed support for opening gzipped FITS files by filename in a writeable mode (PyFITS has supported writing to gzip files for some time now, but only enabled it when GzipFile objects were passed to `pyfits.open()` due to some legacy code preventing full gzip support. (#195)
- Added a more helpful error message in the case of malformed FITS files that contain non-float NULL values in an ASCII table but are missing the required TNULLn keywords in the header. (#197)
- Fixed an (apparently long-standing) issue where writing compressed images did not correctly write the TFORMn keywords for variable-length array columns (they omitted the max array length parameter of the format). (#199)
- Slightly refactored how tables containing variable-length array columns are handled to add two improvements: Fixes an issue where accessing the data after a call to the `pyfits.getdata` convenience function caused an exception, and allows the VLA data to be read from an existing mmap of the FITS file. (#200)
- Fixed a bug that could occur when opening a table containing multi-dimensional columns (i.e. via the TDIMn keyword) and then writing it out to a new file. (#201)
- Added use of the `console_scripts` entry point to install the fitsdiff and fitscheck scripts, which if nothing else provides better Windows support. The generated scripts now override the ones explicitly defined in the `scripts/` directory (which were just trivial stubs to begin with). (#202)
- Fixed a bug on Python 3 where attempting to open a non-existent file on Python 3 caused a seemingly unrelated traceback. (#203)
- Fixed a bug in fitsdiff that reported two header keywords containing NaN as value as different. (#204)
- Fixed an issue in the tests that caused some tests to fail if pyfits is installed with read-only permissions. (#208)
- Fixed a bug where instantiating a BinTableHDU from a numpy array containing boolean fields converted all the values to `False`. (#215)
- Fixed an issue where passing an array of integers into the constructor of `Column()` when the column type is floats of the same byte width caused the column array to become garbled. (#218)

- Fixed inconsistent behavior in creating CONTINUE cards from byte strings versus Unicode strings in Python 2—CONTINUE cards can now be created properly from Unicode strings (so long as they are convertible to ASCII). (spacetelescope/PyFITS#1)
- Fixed a couple cases where creating a new table using TDIMn in some of the columns could caused a crash. (spacetelescope/PyFITS#3)
- Fixed a bug in parsing HIERARCH keywords that do not have a space after the first equals sign (before the value). (spacetelescope/PyFITS#5)
- Prevented extra leading whitespace on HIERARCH keywords from being treated as part of the keyword. (spacetelescope/PyFITS#6)
- Fixed a bug where HIERARCH keywords containing lower-case letters was mistakenly marked as invalid during header validation. (spacetelescope/PyFITS#7)
- Fixed an issue that was ancillary to (spacetelescope/PyFITS#7) where the `Header.index()` method did not work correctly with HIERARCH keywords containing lower-case letters.

3.0.11 (2013-04-17)

- Fixed support for opening gzipped FITS files by filename in a writeable mode (PyFITS has supported writing to gzip files for some time now, but only enabled it when `GzipFile` objects were passed to `pyfits.open()` due to some legacy code preventing full gzip support. Backported from 3.1.2. (#195)
- Added a more helpful error message in the case of malformed FITS files that contain non-float NULL values in an ASCII table but are missing the required TNULLn keywords in the header. Backported from 3.1.2. (#197)
- Fixed an (apparently long-standing) issue where writing compressed images did not correctly write the TFORMn keywords for variable-length array columns (they omitted the max array length parameter of the format). Backported from 3.1.2. (#199)
- Slightly refactored how tables containing variable-length array columns are handled to add two improvements: Fixes an issue where accessing the data after a call to the `pyfits.getdata` convenience function caused an exception, and allows the VLA data to be read from an existing mmap of the FITS file. Backported from 3.1.2. (#200)
- Fixed a bug that could occur when opening a table containing multi-dimensional columns (i.e. via the TDIMn keyword) and then writing it out to a new file. Backported from 3.1.2. (#201)
- Fixed a bug on Python 3 where attempting to open a non-existent file on Python 3 caused a seemingly unrelated traceback. Backported from 3.1.2. (#203)
- Fixed a bug in `fitsdiff` that reported two header keywords containing NaN as value as different. Backported from 3.1.2. (#204)
- Fixed an issue in the tests that caused some tests to fail if `pyfits` is installed with read-only permissions. Backported from 3.1.2. (#208)
- Fixed a bug where instantiating a `BinTableHDU` from a numpy array containing boolean fields converted all the values to `False`. Backported from 3.1.2. (#215)
- Fixed an issue where passing an array of integers into the constructor of `Column()` when the column type is floats of the same byte width caused the column array to become garbled. Backported from 3.1.2. (#218)
- Fixed a couple cases where creating a new table using TDIMn in some of the columns could caused a crash. Backported from 3.1.2. (spacetelescope/PyFITS#3)

3.1.1 (2013-01-02)

This is a bug fix release for the 3.1.x series.

Bug Fixes

- Improved handling of scaled images and pseudo-unsigned integer images in compressed image HDUs. They now work more transparently like normal image HDUs with support for the `do_not_scale_image_data` and `uint` options, as well as `scale_back` and `save_backup`. The `.scale()` method works better too. (#88)
- Permits non-string values for the EXTNAME keyword when reading in a file, rather than throwing an exception due to the malformatting. Added verification for the format of the EXTNAME keyword when writing. (#96)
- Added support for EXTNAME and EXTVER in PRIMARY HDUs. That is, if EXTNAME is specified in the header, it will also be reflected in the `.name` attribute and in `pyfits.info()`. These keywords used to be verboten in PRIMARY HDUs, but the latest version of the FITS standard allows them. (#151)
- HCOMPRESS can again be used to compress data cubes (and higher-dimensional arrays) so long as the tile size is effectively 2-dimensional. In fact, PyFITS will automatically use compatible tile sizes even if they're not explicitly specified. (#171)
- Added support for the optional `endcard` parameter in the `Header.fromtextfile()` and `Header.totextfile()` methods. Although `endcard=False` was a reasonable default assumption, there are still text dumps of FITS headers that include the END card, so this should have been more flexible. (#176)
- Fixed a crash when running `fitsdiff` on two empty (that is, zero row) tables. (#178)
- Fixed an issue where opening files containing random groups HDUs in update mode could cause an unnecessary rewrite of the file even if none of the data is modified. (#179)
- Fixed a bug that could caused a deadlock in the filesystem on OSX if PyFITS is used with Numpy 1.7 in some cases. (#180)
- Fixed a crash when generating diff reports from diffs using the `ignore_comments` options. (#181)
- Fixed some bugs with WCS Paper IV record-valued keyword cards:
 - Cards that looked kind of like RVKCs but were not intended to be were over-permissively treated as such—commentary keywords like COMMENT and HISTORY were particularly affected. (#183)
 - Looking up a card in a header by its standard FITS keyword only should always return the raw value of that card. That way cards containing values that happen to valid RVKCs but were not intended to be will still be treated like normal cards. (#184)
 - Looking up a RVKC in a header with only part of the field-specifier (for example “DPI.AXIS” instead of “DPI.AXIS.1”) was implicitly treated as a wildcard lookup. (#184)
- Fixed a crash when diffing two FITS files where at least one contains a compressed image HDU which was not recognized as an image instead of a table. (#187)
- Fixed bugs in the backwards compatibility layer for the `CardList.index` and `CardList.count` methods. (#190)
- Improved `__repr__` and text file representation of cards with long values that are split into CONTINUE cards. (#193)
- Fixed a crash when trying to assign a long (> 72 character) value to blank (‘’) keywords. This also changed how blank keywords are represented—there are still exactly 8 spaces before any commentary content can begin; this *may* affect the exact display of header cards that assumed there could be fewer spaces in a blank keyword

card before the content begins. However, the current approach is more in line with the requirements of the FITS standard. (#194)

3.0.10 (2013-01-02)

- Improved handling of scaled images and pseudo-unsigned integer images in compressed image HDUs. They now work more transparently like normal image HDUs with support for the `do_not_scale_image_data` and `uint` options, as well as `scale_back` and `save_backup`. The `.scale()` method works better too. Backported from 3.1.1. (#88)
- Permits non-string values for the EXTNAME keyword when reading in a file, rather than throwing an exception due to the malformatting. Added verification for the format of the EXTNAME keyword when writing. Backported from 3.1.1. (#96)
- Added support for EXTNAME and EXTVER in PRIMARY HDUs. That is, if EXTNAME is specified in the header, it will also be reflected in the `.name` attribute and in `pyfits.info()`. These keywords used to be verboten in PRIMARY HDUs, but the latest version of the FITS standard allows them. Backported from 3.1.1. (#151)
- HCOMPRESS can again be used to compress data cubes (and higher-dimensional arrays) so long as the tile size is effectively 2-dimensional. In fact, PyFITS will not automatically use compatible tile sizes even if they're not explicitly specified. Backported from 3.1.1. (#171)
- Fixed a bug when writing out files containing zero-width table columns, where the TFIELDS keyword would be updated incorrectly, leaving the table largely unreadable. Backported from 3.1.0. (#174)
- Fixed an issue where opening files containing random groups HDUs in update mode could cause an unnecessary rewrite of the file even if none of the data is modified. Backported from 3.1.1. (#179)
- Fixed a bug that could caused a deadlock in the filesystem on OSX if PyFITS is used with Numpy 1.7 in some cases. Backported from 3.1.1. (#180)

3.1 (2012-08-08)

Highlights

- The `Header` object has been significantly reworked, and `CardList` objects are now deprecated (their functionality folded into the `Header` class). See API Changes below for more details.
- Memory maps are now used by default to access HDU data. See API Changes below for more details.
- Now includes a new version of the `fitsdiff` program for comparing two FITS files, and a new FITS comparison API used by `fitsdiff`. See New Features below.

API Changes

- The `Header` class has been rewritten, and the `CardList` class is deprecated. Most of the basic details of working with FITS headers are unchanged, and will not be noticed by most users. But there are differences in some areas that will be of interest to advanced users, and to application developers. For full details of the changes, see the “Header Interface Transition Guide” section in the PyFITS documentation. See ticket #64 on the PyFITS Trac for further details and background. Some highlights are listed below:
 - The `Header` class now fully implements the Python dict interface, and can be used interchangeably with a dict, where the keys are header keywords.

- New keywords can be added to the header using normal keyword assignment (previously it was necessary to use `Header.update` to add new keywords). For example:

```
>>> header['NAXIS'] = 2
```

will update the existing 'FOO' keyword if it already exists, or add a new one if it doesn't exist, just like a dict.

- It is possible to assign both a value and a comment at the same time using a tuple:

```
>>> header['NAXIS'] = (2, 'Number of axes')
```

- To add/update a new card and ensure it's added in a specific location, use `Header.set()`:

```
>>> header.set('NAXIS', 2, 'Number of axes', after='BITPIX')
```

This works the same as the old `Header.update()`. `Header.update()` still works in the old way too, but is deprecated.

- Although `Card` objects still exist, it generally is not necessary to work with them directly. `Header.ascardlist()/Header.ascard` are deprecated and should not be used. To directly access the `Card` objects in a header, use `Header.cards`.
- To access card comments, it is still possible to either go through the card itself, or through `Header.comments`. For example:

```
>>> header.cards['NAXIS'].comment
Number of axes
>>> header.comments['NAXIS']
Number of axes
```

- Card objects can now be used interchangeably with `(keyword, value, comment)` 3-tuples. They still have `.value` and `.comment` attributes as well. The `.key` attribute has been renamed to `.keyword` for consistency, though `.key` is still supported (but deprecated).
- Memory mapping is now used by default to access HDU data. That is, `pyfits.open()` uses `mmap=True` as the default. This provides better performance in the majority of use cases—there are only some I/O intensive applications where it might not be desirable. Enabling `mmap` by default also enabled finding and fixing a large number of bugs in PyFITS' handling of memory-mapped data (most of these bug fixes were backported to PyFITS 3.0.5). (#85)
 - A new `pyfits.USE_MEMMAP` global variable was added. Set `pyfits.USE_MEMMAP = False` to change the default `mmap` setting for opening files. This is especially useful for controlling the behavior in applications where `pyfits` is deeply embedded.
 - Likewise, a new `PYFITS_USE_MEMMAP` environment variable is supported. Set `PYFITS_USE_MEMMAP = 0` in your environment to change the default behavior.
- The `size()` method on HDU objects is now a `.size` property—this returns the size in bytes of the data portion of the HDU, and in most cases is equivalent to `hdu.data.nbytes` (#83)
- `BinTableHDU.tdump` and `BinTableHDU.tcreate` are deprecated—use `BinTableHDU.dump` and `BinTableHDU.load` instead. The new methods output the table data in a slightly different format from previous versions, which places quotes around each value. This format is compatible with data dumps from previous versions of PyFITS, but not vice-versa due to a parsing bug in older versions.
- Likewise the `pyfits.tdump` and `pyfits.tcreate` convenience function versions of these methods have been renamed `pyfits.tabledump` and `pyfits.tableload`. The old deprecated, but currently retained for backwards compatibility. (r1125)

- A new global variable `pyfits.EXTENSION_NAME_CASE_SENSITIVE` was added. This serves as a replacement for `pyfits.setExtensionNameCaseSensitive` which is not deprecated and may be removed in a future version. To enable case-sensitivity of extension names (i.e. treat 'sci' as distinct from 'SCI') set `pyfits.EXTENSION_NAME_CASE_SENSITIVE = True`. The default is `False`. (r1139)
- A new global configuration variable `pyfits.STRIP_HEADER_WHITESPACE` was added. By default, if a string value in a header contains trailing whitespace, that whitespace is automatically removed when the value is read. Now if you set `pyfits.STRIP_HEADER_WHITESPACE = False` all whitespace is preserved. (#146)
- The old `classExtensions` extension mechanism (which was deprecated in PyFITS 3.0) is removed outright. To our knowledge it was no longer used anywhere. (r1309)
- Warning messages from PyFITS issued through the Python warnings API are now output to `stderr` instead of `stdout`, as is the default. PyFITS no longer modifies the default behavior of the warnings module with respect to which stream it outputs to. (r1319)
- The `checksum` argument to `pyfits.open()` now accepts a value of 'remove', which causes any existing CHECKSUM/DATASUM keywords to be ignored, and removed when the file is saved.

New Features

- Added support for the proposed "FITS" extension HDU type. See <http://listmgr.cv.nrao.edu/pipermail/fitsbits/2002-April/001094.html>. FITS HDUs contain an entire FITS file embedded in their data section. `FitsHDU` objects work like other HDU types in PyFITS. Their `.data` attribute returns the raw data array. However, they have a special `.hdulist` attribute which processes the data as a FITS file and returns it as an in-memory `HDUList` object. `FitsHDU` objects also support a `FitsHDU.fromhdulist()` classmethod which returns a new `FitsHDU` object that embeds the supplied `HDUList`. (#80)
- Added a new `.is_image` attribute on HDU objects, which is `True` if the HDU data is an 'image' as opposed to a table or something else. Here the meaning of 'image' is fairly loose, and mostly just means a Primary or Image extension HDU, or possibly a compressed image HDU (#71)
- Added an `HDUList.fromstring` classmethod which can parse a FITS file already in memory and instantiate an `HDUList` object from it. This could be useful for integrating PyFITS with other libraries that work on FITS file, such as CFITSIO. It may also be useful in streaming applications. The name is a slight misnomer, in that it actually accepts any Python object that implements the buffer interface, which includes `bytes`, `bytearray`, `memoryview`, `numpy.ndarray`, etc. (#90)
- Added a new `pyfits.diff` module which contains facilities for comparing FITS files. One can use the `pyfits.diff.FITSDiff` class to compare two FITS files in their entirety. There is also a `pyfits.diff.HeaderDiff` class for just comparing two FITS headers, and other similar interfaces. See the PyFITS Documentation for more details on this interface. The `pyfits.diff` module powers the new `fitsdiff` program installed with PyFITS. After installing PyFITS, run `fitsdiff --help` for usage details.
- `pyfits.open()` now accepts a `scale_back` argument. If set to `True`, this automatically scales the data using the original BZERO and BSCALE parameters the file had when it was first opened, if any, as well as the original BITPIX. For example, if the original BITPIX were 16, this would be equivalent to calling `hdu.scale('int16', 'old')` just before calling `flush()` or `close()` on the file. This option applies to all HDUs in the file. (#120)
- `pyfits.open()` now accepts a `save_backup` argument. If set to `True`, this automatically saves a backup of the original file before flushing any changes to it (this of course only applies to update and append mode). This may be especially useful when working with scaled image data. (#121)

Changes in Behavior

- Warnings from PyFITS are not output to stderr by default, instead of stdout as it has been for some time. This is contrary to most users' expectations and makes it more difficult for them to separate output from PyFITS from the desired output for their scripts. (r1319)

Bug Fixes

- Fixed `pyfits.tcreate()` (now `pyfits.tableload()`) to be more robust when encountering blank lines in a column definition file (#14)
- Fixed a fairly rare crash that could occur in the handling of CONTINUE cards when using Numpy 1.4 or lower (though 1.4 is the oldest version supported by PyFITS). (r1330)
- Fixed `_BaseHDU.fromstring` to actually correctly instantiate an HDU object from a string/buffer containing the header and data of that HDU. This allowed for the implementation of `HDUList.fromstring` described above. (#90)
- Fixed a rare corner case where, in some use cases, (mildly, recoverably) malformed float values in headers were not properly returned as floats. (#137)
- Fixed a corollary to the previous bug where float values with a leading zero before the decimal point had the leading zero unnecessarily removed when saving changes to the file (eg. "0.001" would be written back as ".001" even if no changes were otherwise made to the file). (#137)
- When opening a file containing CHECKSUM and/or DATASUM keywords in update mode, the CHECKSUM/DATASUM are updated and preserved even if the file was opened with `checksum=False`. This change in behavior prevents checksums from being unintentionally removed. (#148)
- Fixed a bug where `ImageHDU.scale(option='old')` wasn't working at all—it was not restoring the image to its original BSCALE and BZERO values. (#162)
- Fixed a bug when writing out files containing zero-width table columns, where the TFIELDS keyword would be updated incorrectly, leaving the table largely unreadable. This fix will be backported to the 3.0.x series in version 3.0.10. (#174)

3.0.9 (2012-08-06)

This is a bug fix release for the 3.0.x series.

Bug Fixes

- Fixed `Header.values()/Header.itervalues()` and `Header.items()/Header.iteritems()` to correctly return the different values for duplicate keywords (particularly commentary keywords like HISTORY and COMMENT). This makes the old Header implementation slightly more compatible with the new implementation in PyFITS 3.1. (#127)

Note: This fix did not change the existing behavior from earlier PyFITS versions where `Header.keys()` returns all keywords in the header with duplicates removed. PyFITS 3.1 changes that behavior, so that `Header.keys()` includes duplicates.

- Fixed a bug where `ImageHDU.scale(option='old')` wasn't working at all—it was not restoring the image to its original BSCALE and BZERO values. (#162)

- Fixed a bug where opening a file containing compressed image HDUs in ‘update’ mode and then immediately closing it without making any changes caused the file to be rewritten unnecessarily. (#167)
- Fixed two memory leaks that could occur when writing compressed image data, or in some cases when opening files containing compressed image HDUs in ‘update’ mode. (#168)

3.0.8 (2012-06-04)

Changes in Behavior

- Prior to this release, image data sections did not work with scaled data—that is, images with non-trivial BSCALE and/or BZERO values. Previously, in order to read such images in sections, it was necessary to manually apply the BSCALE+BZERO to each section. It’s worth noting that sections *did* support pseudo-unsigned ints (flakily). This change just extends that support for general BSCALE+BZERO values.

Bug Fixes

- Fixed a bug that prevented updates to values in boolean table columns from being saved. This turned out to be a symptom of a deeper problem that could prevent other table updates from being saved as well. (#139)
- Fixed a corner case in which a keyword comment ending with the string “END” could, in some circumstances, cause headers (and the rest of the file after that point) to be misread. (#142)
- Fixed support for scaled image data and psuedo-unsigned ints in image data sections (`hdu.section`). Previously this was not supported at all. At some point support was supposedly added, but it was buggy and incomplete. Now the feature seems to work much better. (#143)
- Fixed the documentation to point out that image data sections *do* support non-contiguous slices (and have for a long time). The documentation was never updated to reflect this, and misinformed users that only contiguous slices were supported, leading to some confusion. (#144)
- Fixed a bug where creating an `HDUList` object containing multiple PRIMARY HDUs caused an infinite recursion when validating the object prior to writing to a file. (#145)
- Fixed a rare but serious case where saving an update to a file that previously had a CHECKSUM and/or DATA-SUM keyword, but removed the checksum in saving, could cause the file to be slightly corrupted and unreadable. (#147)
- Fixed problems with reading “non-standard” FITS files with primary headers containing `SIMPLE = F`. PyFITS has never made many guarantees as to how such files are handled. But it should at least be possible to read their headers, and the data if possible. Saving changes to such a file should not try to prepend an unwanted valid PRIMARY HDU. (#157)
- Fixed a bug where opening an image with `disable_image_compression = True` caused compression to be disabled for all subsequent `pyfits.open()` calls. (r1651)

3.0.7 (2012-04-10)

Changes in Behavior

- Slices of `GroupData` objects now return new `GroupData` objects instead of extended multi-row `_Group` objects. This is analogous to how PyFITS 3.0 fixed `FITS_rec` slicing, and should have been fixed for `GroupData` at the same time. The old behavior caused bugs where functions internal to Numpy expected that slicing an ndarray would return a new ndarray. As this is a rare usecase with a rare feature most users are unlikely to be affected by this change.

- The previously internal `_Group` object for representing individual group records in a `GroupData` object are renamed `Group` and are now a public interface. However, there's almost no good reason to create `Group` objects directly, so it shouldn't be considered a "new feature".
- An annoyance from PyFITS 3.0.6 was fixed, where the value of the `EXTEND` keyword was always being set to `F` if there are not actually any extension HDUs. It was unnecessary to modify this value.

Bug Fixes

- Fixed `GroupData` objects to return new `GroupData` objects when sliced instead of `_Group` record objects. See "Changes in behavior" above for more details.
- Fixed slicing of `Group` objects—previously it was not possible to slice slice them at all.
- Made it possible to assign `np.bool_` objects as header values. (#123)
- Fixed overly strict handling of the `EXTEND` keyword; see "Changes in behavior" above. (#124)
- Fixed many cases where an HDU's header would be marked as "modified" by PyFITS and rewritten, even when no changes to the header are necessary. (#125)
- Fixed a bug where the values of the `PTYPEn` keywords in a random groups HDU were forced to be all lower-case when saving the file. (#130)
- Removed an unnecessary inline import in `ExtensionHDU.__setattr__` that was causing some slowdown when opening files containing a large number of extensions, plus a few other small (but not insignificant) performance improvements thanks to Julian Taylor. (#133)
- Fixed a regression where header blocks containing invalid end-of-header padding (i.e. null bytes instead of spaces) couldn't be parsed by PyFITS. Such headers can be parsed again, but a warning is raised, as such headers are not valid FITS. (#136)
- Fixed a memory leak where table data in random groups HDUs weren't being garbage collected. (#138)

3.0.6 (2012-02-29)

Highlights

The main reason for this release is to fix an issue that was introduced in PyFITS 3.0.5 where merely opening a file containing scaled data (that is, with non-trivial `BSCALE` and `BZERO` keywords) in 'update' mode would cause the data to be automatically rescaled—possibly converting the data from ints to floats—as soon as the file is closed, even if the application did not touch the data. Now PyFITS will only rescale the data in an extension when the data is actually accessed by the application. So opening a file in 'update' mode in order to modify the header or append new extensions will not cause any change to the data in existing extensions.

This release also fixes a few Windows-specific bugs found through more extensive Windows testing, and other miscellaneous bugs.

Bug Fixes

- More accurate error messages when opening files containing invalid header cards. (#109)
- Fixed a possible reference cycle/memory leak that was caught through more extensive testing on Windows. (#112)
- Fixed 'ostream' mode to open the underlying file in 'wb' mode instead of 'w' mode. (#112)

- Fixed a Windows-only issue where trying to save updates to a resized FITS file could result in a crash due to there being open mmmaps on that file. (#112)
- Fixed a crash when trying to create a FITS table (i.e. with `new_table()`) from a Numpy array containing bool fields. (#113)
- Fixed a bug where manually initializing an `HDUList` with a list of HDUs wouldn't set the correct `EXTEND` keyword value on the primary HDU. (#114)
- Fixed a crash that could occur when trying to `deepcopy` a `Header` in Python < 2.7. (#115)
- Fixed an issue where merely opening a scaled image in 'update' mode would cause the data to be converted to floats when the file is closed. (#119)

3.0.5 (2012-01-30)

- Fixed a crash that could occur when accessing image sections of files opened with `memmap=True`. (r1211)
- Fixed the inconsistency in the behavior of files opened in 'readonly' mode when `memmap=True` vs. when `memmap=False`. In the latter case, although changes to array data were not saved to disk, it was possible to update the array data in memory. On the other hand with `memmap=True`, 'readonly' mode prevented even in-memory modification to the data. This is what 'copyonwrite' mode was for, but difference in behavior was confusing. Now 'readonly' is equivalent to 'copyonwrite' when using `memmap`. If the old behavior of denying changes to the array data is necessary, a new 'denywrite' mode may be used, though it is only applicable to files opened with `memmap`. (r1275)
- Fixed an issue where files opened with `memmap=True` would return image data as a raw `numpy.memmap` object, which can cause some unexpected behaviors—instead `memmap` object is viewed as a `numpy.ndarray`. (r1285)
- Fixed an issue in Python 3 where a workaround for a bug in Numpy on Python 3 interacted badly with some other software, namely to `vo.table` package (and possibly others). (r1320, r1337, and #110)
- Fixed buggy behavior in the handling of SIGINTs (i.e. Ctrl-C keyboard interrupts) while flushing changes to a FITS file. PyFITS already prevented SIGINTs from causing an incomplete flush, but did not clean up the signal handlers properly afterwards, or reraise the keyboard interrupt once the flush was complete. (r1321)
- Fixed a crash that could occur in Python 3 when opening files with checksum checking enabled. (r1336)
- Fixed a small bug that could cause a crash in the `StreamingHDU` interface when using Numpy below version 1.5.
- Fixed a crash that could occur when creating a new `CompImageHDU` from an array of big-endian data. (#104)
- Fixed a crash when opening a file with extra zero padding at the end. Though FITS files should not have such padding, it's not explicitly forbidden by the format either, and PyFITS shouldn't stumble over it. (#106)
- Fixed a major slowdown in opening tables containing large columns of string values. (#111)

3.0.4 (2011-11-22)

- Fixed a crash when writing HCOMPRESS compressed images that could happen on Python 2.5 and 2.6. (r1217)
- Fixed a crash when slicing an table in a file opened in 'readonly' mode with `memmap=True`. (r1230)
- Writing changes to a file or writing to a new file verifies the output in 'fix' mode by default instead of 'exception'—that is, PyFITS will automatically fix common FITS format errors rather than raising an exception. (r1243)
- Fixed a bug where convenience functions such as `getval()` and `getheader()` crashed when specifying just 'PRIMARY' as the extension to use (r1263).

- Fixed a bug that prevented passing keyword arguments (beyond the standard data and header arguments) as positional arguments to the constructors of extension HDU classes.
- Fixed some tests that were failing on Windows—in this case the tests themselves failed to close some temp files and Windows refused to delete them while there were still open handles on them. (r1295)
- Fixed an issue with floating point formatting in header values on Python 2.5 for Windows (and possibly other platforms). The exponent was zero-padded to 3 digits; although the FITS standard makes no specification on this, the formatting is now normalized to always pad the exponent to two digits. (r1295)
- Fixed a bug where long commentary cards (such as HISTORY and COMMENT) were broken into multiple CONTINUE cards. However, commentary cards are not expected to be found in CONTINUE cards. Instead these long cards are broken into multiple commentary cards. (#97)
- GZIP/ZIP-compressed FITS files can be detected and opened regardless of their filename extension. (#99)
- Fixed a serious bug where opening scaled images in ‘update’ mode and then closing the file without touching the data would cause the file to be corrupted. (#101)

3.0.3 (2011-10-05)

- Fixed several small bugs involving corner cases in record-valued keyword cards (#70)
- In some cases HDU creation failed if the first keyword value in the header was not a string value (#89)
- Fixed a crash when trying to compute the HDU checksum when the data array contains an odd number of bytes (#91)
- Disabled an unnecessary warning that was displayed on opening compressed HDUs with `disable_image_compression = True` (#92)
- Fixed a typo in code for handling HCOMPRESS compressed images.

3.0.2 (2011-09-23)

- The `BinTableHDU.tcreate` method and by extension the `pyfits.tcreate` function don’t get tripped up by blank lines anymore (#14)
- The presence, value, and position of the EXTEND keyword in Primary HDUs is verified when reading/writing a FITS file (#32)
- Improved documentation (in warning messages as well as in the handbook) that PyFITS uses zero-based indexing (as one would expect for C/Python code, but contrary to the PyFITS standard which was written with FORTRAN in mind) (#68)
- Fixed a bug where updating a header card comment could cause the value to be lost if it had not already been read from the card image string.
- Fixed a related bug where changes made directly to Card object in a header (i.e. assigning directly to `card.value` or `card.comment`) would not propagate when flushing changes to the file (#69) [Note: This and the bug above it were originally reported as being fixed in version 3.0.1, but the fix was never included in the release.]
- Improved file handling, particularly in Python 3 which had a few small file I/O-related bugs (#76)
- Fixed a bug where updating a FITS file would sometimes cause it to lose its original file permissions (#79)
- Fixed the handling of TDIMn keywords; 3.0 added support for them, but got the axis order backwards (they were treated as though they were row-major) (#82)
- Fixed a crash when a FITS file containing scaled data is opened and immediately written to a new file without explicitly viewing the data first (#84)

- Fixed a bug where creating a table with columns named either ‘names’ or ‘formats’ resulted in an infinite recursion (#86)

3.0.1 (2011-09-12)

- Fixed a bug where updating a header card comment could cause the value to be lost if it had not already been read from the card image string.
- Changed `_TableBaseHDU.data` so that if the data contain an empty table a `FITS_rec` object with zero rows is returned rather than `None`.
- The `.key` attribute of `RecordValuedKeywordCards` now returns the full keyword+field-specifier value, instead of just the plain keyword (#46)
- Fixed a related bug where changes made directly to `Card` object in a header (i.e. assigning directly to `card.value` or `card.comment`) would not propagate when flushing changes to the file (#69)
- Fixed a bug where writing a table with zero rows could fail in some cases (#72)
- Miscellaneous small bug fixes that were causing some tests to fail, particularly on Python 3 (#74, #75)
- Fixed a bug where creating a table column from an array in non-native byte order would not preserve the byte order, thus interpreting the column array using the wrong byte order (#77)

3.0.0 (2011-08-23)

- Contains major changes, bumping the version to 3.0
- Large amounts of refactoring and reorganization of the code; tried to preserve public API backwards-compatibility with older versions (private API has many changes and is not guaranteed to be backwards-compatible). There are a few small public API changes to be aware of:
 - The `pyfits.rec` module has been removed completely. If your version of `numpy` does not have the `numpy.core.records` module it is too old to be used with `PyFITS`.
 - The `Header.ascardlist()` method is deprecated—use the `.ascard` attribute instead.
 - `Card` instances have a new `.cardimage` attribute that should be used rather than `.ascardimage()`, which may become deprecated.
 - The `Card.fromstring()` method is now a classmethod. It returns a new `Card` instance rather than modifying an existing instance.
 - The `req_cards()` method on `HDU` instances has changed: The `pos` argument is not longer a string. It is either an integer value (meaning the card’s position must match that value) or it can be a function that takes the card’s position as it’s argument, and returns `True` if the position is valid. Likewise, the `test` argument no longer takes a string, but instead a function that validates the card’s value and returns `True` or `False`.
 - The `get_coldefs()` method of table `HDU`s is deprecated. Use the `.columns` attribute instead.
 - The `ColDefs.data` attribute is deprecated—use `ColDefs.columns` instead (though in general you shouldn’t mess with it directly—it might become internal at some point).
 - `FITS_record` objects take `start` and `end` as arguments instead of `startColumn` and `endColumn` (these are rarely created manually, so it’s unlikely that this change will affect anyone).
 - `BinTableHDU.tcreate()` is now a classmethod, and returns a new `BinTableHDU` instance.
 - Use `ExtensionHDU` and `NonstandardExtHDU` for making new extension `HDU` classes. They are now public interfaces, whereas previously they were private and prefixed with underscores.

– Possibly others—please report if you find any changes that cause difficulties.

- Calls to deprecated functions will display a Deprecation warning. However, in Python 2.7 and up Deprecation warnings are ignored by default, so run Python with the `-Wd` option to see if you're using any deprecated functions. If we get close to actually removing any functions, we might make the Deprecation warnings display by default.
- Added basic Python 3 support
- Added support for multi-dimensional columns in tables as specified by the `TDIMn` keywords (#47)
- Fixed a major memory leak that occurred when creating new tables with the `new_table()` function (#49) be padded with zero-bytes) vs ASCII tables (where strings are padded with spaces) (#15)
- Fixed a bug in which the case of Random Access Group parameters names was not preserved when writing (#41)
- Added support for binary table fields with zero width (#42)
- Added support for wider integer types in ASCII tables; although this is non- standard, some GEIS images require it (#45)
- Fixed a bug that caused the `index_of()` method of HDULists to crash when the HDUList object is created from scratch (#48)
- Fixed the behavior of string padding in binary tables (where strings should be padded with nulls instead of spaces)
- Fixed a rare issue that caused excessive memory usage when computing checksums using a non-standard block size (see r818)
- Add support for forced uint data in image sections (#53)
- Fixed an issue where variable-length array columns were not extended when creating a new table with more rows than the original (#54)
- Fixed tuple and list-based indexing of `FITS_rec` objects (#55)
- Fixed an issue where `BZERO` and `BSCALE` keywords were appended to headers in the wrong location (#56)
- `FITS_record` objects (table rows) have full slicing support, including stepping, etc. (#59)
- Fixed a bug where updating multiple files simultaneously (such as when running parallel processes) could lead to a race condition with `mktemp()` (#61)
- Fixed a bug where compressed image headers were not in the order expected by the `funpack` utility (#62)

2.4.0 (2011-01-10)

The following enhancements were added:

- Checksum support now correctly conforms to the FITS standard. `pyfits` supports reading and writing both the old checksums and new standard-compliant checksums. The `fitscheck` command-line utility is provided to verify and update checksums.
- Added a new optional keyword argument `do_not_scale_image_data` to the `pyfits.open` convenience function. When this argument is provided as `True`, and an `ImageHDU` is read that contains scaled data, the data is not automatically scaled when it is read. This option may be used when opening a fits file for update, when you only want to update some header data. Without the use of this argument, if the header updates required the size of the fits file to change, then when writing the updated information, the data would be read, scaled, and written back out in its scaled format (usually with a different data type) instead of in its non-scaled format.

- Added a new optional keyword argument `disable_image_compression` to the `pyfits.open` function. When `True`, any compressed image HDU's will be read in like they are binary table HDU's.
- Added a `verify` keyword argument to the `pyfits.append` function. When `False`, `append` will assume the existing FITS file is already valid and simply append new content to the end of the file, resulting in a large speed up appending to large files.
- Added HDU methods `update_ext_name` and `update_ext_version` for updating the name and version of an HDU.
- Added HDU method `filebytes` to calculate the number of bytes that will be written to the file associated with the HDU.
- Enhanced the section class to allow reading non-contiguous image data. Previously, the section class could only be used to read contiguous data. (CNSHD781626)
- Added method `HDUList.fileinfo()` that returns a dictionary with information about the location of header and data in the file associated with the HDU.

The following bugs were fixed:

- Reading in some malformed FITS headers would cause a `NameError` exception, rather than information about the cause of the error.
- `pyfits` can now handle non-compliant `CONTINUE` cards produced by Java FITS.
- `BinTable` columns with `TSCALn` are now byte-swapped correctly.
- Ensure that floating-point card values are no longer than 20 characters.
- Updated `flush` so that when the data has changed in an HDU for a file opened in update mode, the header will be updated to match the changed data before writing out the HDU.
- Allow `HIERARCH` cards to contain a keyword and value whose total character length is 69 characters. Previous length was limited at 68 characters.
- Calls to `FITS_rec['columnName']` now return an `ndarray`. exactly the same as a call to `FITS_rec.field('columnName')` or `FITS_rec.columnName`. Previously, `FITS_rec['columnName']` returned a much less useful `fits_record` object. (CNSHD789053)
- Corrected the `append` convenience function to eliminate the reading of the HDU data from the file that is being appended to. (CNSHD794738)
- Eliminated common symbols between the `pyfitsComp` module and the `cfitsio` and `zlib` libraries. These can cause problems on systems that use both `PyFITS` and `cfitsio` or `zlib`. (CNSHD795046)

2.3.1 (2010-06-03)

The following bugs were fixed:

- Replaced code in the Compressed Image HDU extension which was covered under a GNU General Public License with code that is covered under a BSD License. This change allows the distribution of `pyfits` under a BSD License.

2.3 (2010-05-11)

The following enhancements were made:

- Completely eliminate support for `numarray`.
- Rework `pyfits` documentation to use Sphinx.

- Support python 2.6 and future division.
- Added a new method to get the file name associated with an HDUList object. The method HDUList.filename() returns the name of an associated file. It returns None if no file is associated with the HDUList.
- Support the python 2.5 'with' statement when opening fits files. (CNSHD766308) It is now possible to use the following construct:

```
>>> from __future__ import with_statement import pyfits
>>> with pyfits.open("input.fits") as hdul:
...     #process hdul
>>>
```

- Extended the support for reading unsigned integer 16 values from an ImageHDU to include unsigned integer 32 and unsigned integer 64 values. ImageHDU data is considered to be unsigned integer 16 when the data type is signed integer 16 and BZERO is equal to 2^{15} (32784) and BSCALE is equal to 1. ImageHDU data is considered to be unsigned integer 32 when the data type is signed integer 32 and BZERO is equal to 2^{31} and BSCALE is equal to 1. ImageHDU data is considered to be unsigned integer 64 when the data type is signed integer 64 and BZERO is equal to 2^{63} and BSCALE is equal to 1. An optional keyword argument (uint) was added to the open convenience function for this purpose. Supplying a value of True for this argument will cause data of any of these types to be read in and scaled into the appropriate unsigned integer array (uint16, uint32, or uint64) instead of into the normal float 32 or float 64 array. If an HDU associated with a file that was opened with the 'int' option and containing unsigned integer 16, 32, or 64 data is written to a file, the data will be reverse scaled into a signed integer 16, 32, or 64 array and written out to the file along with the appropriate BSCALE/BZERO header cards. Note that for backward compatibility, the 'uint16' keyword argument will still be accepted in the open function when handling unsigned integer 16 conversion.
- Provided the capability to access the data for a column of a fits table by indexing the table using the column name. This is consistent with Record Arrays in numpy (array with fields). (CNSHD763378) The following example will illustrate this:

```
>>> import pyfits
>>> hdul = pyfits.open('input.fits')
>>> table = hdul[1].data
>>> table.names
['c1', 'c2', 'c3', 'c4']
>>> print table.field('c2') # this is the data for column 2
['abc' 'xy']
>>> print table['c2'] # this is also the data for column 2
array(['abc', 'xy'], dtype='|S3')
>>> print table[1] # this is the data for row 1
(2, 'xy', 6.69999997138977054, True)
```

- Provided capabilities to create a BinaryTableHDU directly from a numpy Record Array (array with fields). The new capabilities include table creation, writing a numpy Record Array directly to a fits file using the pyfits.writeto and pyfits.append convenience functions. Reading the data for a BinaryTableHDU from a fits file directly into a numpy Record Array using the pyfits.getdata convenience function. (CNSHD749034) Thanks to Erin Sheldon at Brookhaven National Laboratory for help with this.

The following should illustrate these new capabilities:

```
>>> import pyfits
>>> import numpy
>>> t=numpy.zeros(5, dtype=[('x', 'f4'), ('y', '2i4')]) \
... # Create a numpy Record Array with fields
>>> hdu = pyfits.BinTableHDU(t) \
... # Create a Binary Table HDU directly from the Record Array
>>> print hdu.data
[(0.0, array([0, 0], dtype=int32))]
```

```

(0.0, array([0, 0], dtype=int32))
(0.0, array([0, 0], dtype=int32))
(0.0, array([0, 0], dtype=int32))
(0.0, array([0, 0], dtype=int32))]
>>> hdu.writeto('test1.fits', clobber=True) \
... # Write the HDU to a file
>>> pyfits.info('test1.fits')
Filename: test1.fits
No.    Name          Type          Cards  Dimensions  Format
0     PRIMARY      PrimaryHDU     4      ()          uint8
1           BinTableHDU  12    5R x 2C    [E, 2J]
>>> pyfits.writeto('test.fits', t, clobber=True) \
... # Write the Record Array directly to a file
>>> pyfits.append('test.fits', t) \
... # Append another Record Array to the file
>>> pyfits.info('test.fits')
Filename: test.fits
No.    Name          Type          Cards  Dimensions  Format
0     PRIMARY      PrimaryHDU     4      ()          uint8
1           BinTableHDU  12    5R x 2C    [E, 2J]
2           BinTableHDU  12    5R x 2C    [E, 2J]
>>> d=pyfits.getdata('test.fits', ext=1) \
... # Get the first extension from the file as a FITS_rec
>>> print type(d)
<class 'pyfits.core.FITS_rec'>
>>> print d
[(0.0, array([0, 0], dtype=int32))
 (0.0, array([0, 0], dtype=int32))
 (0.0, array([0, 0], dtype=int32))
 (0.0, array([0, 0], dtype=int32))
 (0.0, array([0, 0], dtype=int32))]
>>> d=pyfits.getdata('test.fits', ext=1, view=np.ndarray) \
... # Get the first extension from the file as a numpy Record
    Array
>>> print type(d)
<type 'numpy.ndarray'>
>>> print d
[(0.0, [0, 0]) (0.0, [0, 0]) (0.0, [0, 0]) (0.0, [0, 0])
 (0.0, [0, 0])]
>>> print d.dtype
[('x', '>f4'), ('y', '>i4', 2)]
>>> d=pyfits.getdata('test.fits', ext=1, upper=True,
...                  view=pyfits.FITS_rec) \
... # Force the Record Array field names to be in upper case
    regardless of how they are stored in the file
>>> print d.dtype
[('X', '>f4'), ('Y', '>i4', 2)]

```

- Provided support for writing fits data to file-like objects that do not support the random access methods seek() and tell(). Most pyfits functions or methods will treat these file-like objects as an empty file that cannot be read, only written. It is also expected that the file-like object is in a writable condition (ie. opened) when passed into a pyfits function or method. The following methods and functions will allow writing to a non-random access file-like object: HDUList.writeto(), HDUList.flush(), pyfits.writeto(), and pyfits.append(). The pyfits.open() convenience function may be used to create an HDUList object that is associated with the provided file-like object. (CNSHD770036)

An illustration of the new capabilities follows. In this example fits data is written to standard output which is associated with a file opened in write-only mode:

```

>>> import pyfits
>>> import numpy as np
>>> import sys
>>>
>>> hdu = pyfits.PrimaryHDU(np.arange(100, dtype=np.int32))
>>> hdul = pyfits.HDUList()
>>> hdul.append(hdu)
>>> tmpfile = open('tmpfile.py', 'w')
>>> sys.stdout = tmpfile
>>> hdul.writeto(sys.stdout, clobber=True)
>>> sys.stdout = sys.__stdout__
>>> tmpfile.close()
>>> pyfits.info('tmpfile.py')
Filename: tmpfile.py
No.      Name      Type      Cards   Dimensions   Format
0       PRIMARY    PrimaryHDU    5   (100,)       int32
>>>

```

- Provided support for slicing a `FITS_record` object. The `FITS_record` object represents the data from a row of a table. Pyfits now supports the slice syntax to retrieve values from the row. The following illustrates this new syntax:

```

>>> hdul = pyfits.open('table.fits')
>>> row = hdul[1].data[0]
>>> row
('clear', 'nicmos', 1, 30, 'clear', 'idno= 100')
>>> a, b, c, d, e = row[0:5]
>>> a
'clear'
>>> b
'nicmos'
>>> c
1
>>> d
30
>>> e
'clear'
>>>

```

- Allow the assignment of a row value for a pyfits table using a tuple or a list as input. The following example illustrates this new feature:

```

>>> c1=pyfits.Column(name='target', format='10A')
>>> c2=pyfits.Column(name='counts', format='J', unit='DN')
>>> c3=pyfits.Column(name='notes', format='A10')
>>> c4=pyfits.Column(name='spectrum', format='5E')
>>> c5=pyfits.Column(name='flag', format='L')
>>> coldefs=pyfits.ColDefs([c1,c2,c3,c4,c5])
>>>
>>> tbhdu=pyfits.new_table(coldefs, nrows = 5)
>>>
>>> # Assigning data to a table's row using a tuple
>>> tbhdu.data[2] = ('NGC1', 312, 'A Note',
... num.array([1.1, 2.2, 3.3, 4.4, 5.5], dtype=num.float32),
... True)
>>>
>>> # Assigning data to a tables row using a list
>>> tbhdu.data[3] = ['JIM1', '33', 'A Note',
... num.array([1., 2., 3., 4., 5.], dtype=num.float32), True]

```

- Allow the creation of a Variable Length Format (P format) column from a list of data. The following example illustrates this new feature:

```
>>> a = [num.array([7.2e-20, 7.3e-20]), num.array([0.0]),
... num.array([0.0])]
>>> acol = pyfits.Column(name='testa', format='PD()', array=a)
>>> acol.array
_VLF([[ 7.20000000e-20  7.30000000e-20], [ 0.], [ 0.]],
dtype=object)
>>>
```

- Allow the assignment of multiple rows in a table using the slice syntax. The following example illustrates this new feature:

```
>>> counts = num.array([312, 334, 308, 317])
>>> names = num.array(['NGC1', 'NGC2', 'NGC3', 'NGC4'])
>>> c1=pyfits.Column(name='target', format='10A', array=names)
>>> c2=pyfits.Column(name='counts', format='J', unit='DN',
... array=counts)
>>> c3=pyfits.Column(name='notes', format='A10')
>>> c4=pyfits.Column(name='spectrum', format='5E')
>>> c5=pyfits.Column(name='flag', format='L', array=[1, 0, 1, 1])
>>> coldefs=pyfits.ColDefs([c1, c2, c3, c4, c5])
>>>
>>> tbhdul=pyfits.new_table(coldefs)
>>>
>>> counts = num.array([112, 134, 108, 117])
>>> names = num.array(['NGC5', 'NGC6', 'NGC7', 'NGC8'])
>>> c1=pyfits.Column(name='target', format='10A', array=names)
>>> c2=pyfits.Column(name='counts', format='J', unit='DN',
... array=counts)
>>> c3=pyfits.Column(name='notes', format='A10')
>>> c4=pyfits.Column(name='spectrum', format='5E')
>>> c5=pyfits.Column(name='flag', format='L', array=[0, 1, 0, 0])
>>> coldefs=pyfits.ColDefs([c1, c2, c3, c4, c5])
>>>
>>> tbhdu=pyfits.new_table(coldefs)
>>> tbhdu.data[0][3] = num.array([1., 2., 3., 4., 5.],
... dtype=num.float32)
>>>
>>> tbhdu2=pyfits.new_table(tbhdu1.data, nrows=9)
>>>
>>> # Assign the 4 rows from the second table to rows 5 thru
... 8 of the new table. Note that the last row of the new
... table will still be initialized to the default values.
>>> tbhdu2.data[4:] = tbhdu.data
>>>
>>> print tbhdu2.data
[ ('NGC1', 312, '0.0', array([ 0.,  0.,  0.,  0.,  0.],
dtype=float32), True)
 ('NGC2', 334, '0.0', array([ 0.,  0.,  0.,  0.,  0.],
dtype=float32), False)
 ('NGC3', 308, '0.0', array([ 0.,  0.,  0.,  0.,  0.],
dtype=float32), True)
 ('NGC4', 317, '0.0', array([ 0.,  0.,  0.,  0.,  0.],
dtype=float32), True)
 ('NGC5', 112, '0.0', array([ 1.,  2.,  3.,  4.,  5.],
dtype=float32), False)
 ('NGC6', 134, '0.0', array([ 0.,  0.,  0.,  0.,  0.],
```

```

dtype=float32), True)
    ('NGC7', 108, '0.0', array([ 0.,  0.,  0.,  0.,  0.],
dtype=float32), False)
    ('NGC8', 117, '0.0', array([ 0.,  0.,  0.,  0.,  0.],
dtype=float32), False)
    ('0.0', 0, '0.0', array([ 0.,  0.,  0.,  0.,  0.],
dtype=float32), False)]
>>>

```

The following bugs were fixed:

- Corrected bugs in `HDUList.append` and `HDUList.insert` to correctly handle the situation where you want to insert or append a Primary HDU as something other than the first HDU in an `HDUList` and the situation where you want to insert or append an Extension HDU as the first HDU in an `HDUList`.
- Corrected a bug involving scaled images (both compressed and not compressed) that include a BLANK, or ZBLANK card in the header. When the image values match the BLANK or ZBLANK value, the value should be replaced with NaN after scaling. Instead, `pyfits` was scaling the BLANK or ZBLANK value and returning it. (CNSHD766129)
- Corrected a byteswapping bug that occurs when writing certain column data. (CNSHD763307)
- Corrected a bug that occurs when creating a column from a chararray when one or more elements are shorter than the specified format length. The bug wrote nulls instead of spaces to the file. (CNSHD695419)
- Corrected a bug in the HDU verification software to ensure that the header contains no NAXISn cards where $n > \text{NAXIS}$.
- Corrected a bug involving reading and writing compressed image data. When written, the header keyword card ZTENSION will always have the value 'IMAGE' and when read, if the ZTENSION value is not 'IMAGE' the user will receive a warning, but the data will still be treated as image data.
- Corrected a bug that restricted the ability to create a custom HDU class and use it with `pyfits`. The bug fix will allow something like this:

```

>>> import pyfits
>>> class MyPrimaryHDU(pyfits.PrimaryHDU):
...     def __init__(self, data=None, header=None):
...         pyfits.PrimaryHDU.__init__(self, data, header)
...     def _summary(self):
...         """
...         Reimplement a method of the class.
...         """
...         s = pyfits.PrimaryHDU._summary(self)
...         # change the behavior to suit me.
...         s1 = 'MyPRIMARY ' + s[11:]
...         return s1
...
>>> hdul=pyfits.open("pix.fits",
... classExtensions={pyfits.PrimaryHDU: MyPrimaryHDU})
>>> hdul.info()
Filename: pix.fits
No.      Name          Type          Cards   Dimensions   Format
0       MyPRIMARY    MyPrimaryHDU    59     (512, 512)   int16
>>>

```

- Modified `ColDefs.add_col` so that instead of returning a new `ColDefs` object with the column added to the end, it simply appends the new column to the current `ColDefs` object in place. (CNSHD768778)
- Corrected a bug in `ColDefs.del_col` which raised a `KeyError` exception when deleting a column from a `ColDefs` object.

- Modified the open convenience function so that when a file is opened in readonly mode and the file contains no HDU's an IOError is raised.
- Modified `_TableBaseHDU` to ensure that all locations where data is referenced in the object actually reference the same ndarray, instead of copies of the array.
- Corrected a bug in the Column class that failed to initialize data when the data is a boolean array. (CNSHD779136)
- Corrected a bug that caused an exception to be raised when creating a variable length format column from character data (PA format).
- Modified installation code so that when installing on Windows, when a C++ compiler compatible with the Python binary is not found, the installation completes with a warning that all optional extension modules failed to build. Previously, an Error was issued and the installation stopped.

2.2.2 (2009-10-12)

Updates described in this release are only supported in the NUMPY version of pyfits.

The following bugs were fixed:

- Corrected a bug that caused an exception to be raised when creating a `CompImageHDU` using an initial header that does not match the image data in terms of the number of axis.

2.2.1 (2009-10-06)

Updates described in this release are only supported in the NUMPY version of pyfits.

The following bugs were fixed:

- Corrected a bug that prevented the opening of a fits file where a header contained a CHECKSUM card but no DATASUM card.
- Corrected a bug that caused NULLs to be written instead of blanks when an ASCII table was created using a numpy chararray in which the original data contained trailing blanks. (CNSHD695419)

2.2 (2009-09-23)

Updates described in this release are only supported in the NUMPY version of pyfits.

The following enhancements were made:

- Provide support for the FITS Checksum Keyword Convention. (CNSHD754301)
- Adding the `checksum=True` keyword argument to the open convenience function will cause checksums to be verified on file open:

```
>>> hdul=pyfits.open('in.fits', checksum=True)
```

- On output, CHECKSUM and DATASUM cards may be output to all HDU's in a fits file by using the keyword argument `checksum=True` in calls to the `writeto` convenience function, the `HDUList.writeto` method, the `writeto` methods of all of the HDU classes, and the `append` convenience function:

```
>>> hdul.writeto('out.fits', checksum=True)
```

- Implemented a new `insert` method to the `HDUList` class that allows for the insertion of a HDU into a `HDUList` at a given index:

```
>>> hdul.insert(2, hdu)
```

- Provided the capability to handle Unicode input for file names.
- Provided support for integer division required by Python 3.0.

The following bugs were fixed:

- Corrected a bug that caused an index out of bounds exception to be raised when iterating over the rows of a binary table HDU using the syntax “for row in tbhdu.data: ”. (CNSHD748609)
- Corrected a bug that prevented the use of the writeto convenience function for writing table data to a file. (CNSHD749024)
- Modified the code to raise an IOError exception with the comment “Header missing END card.” when pyfits can’t find a valid END card for a header when opening a file.
 - This change addressed a problem with a non-standard fits file that contained several new-line characters at the end of each header and at the end of the file. However, since some people want to be able to open these non-standard files anyway, an option was added to the open convenience function to allow these files to be opened without exception:

```
>>> pyfits.open('infile.fits', ignore_missing_end=True)
```

- Corrected a bug that prevented the use of StringIO objects as fits files when reading and writing table data. Previously, only image data was supported. (CNSHD753698)
- Corrected a bug that caused a bus error to be generated when compressing image data using GZIP_1 under the Solaris operating system.
- Corrected bugs that prevented pyfits from properly reading Random Groups HDU’s using numpy. (CNSHD756570)
- Corrected a bug that can occur when writing a fits file. (CNSHD757508)
 - If no default SIGINT signal handler has not been assigned, before the write, a TypeError exception is raised in the _File.flush() method when attempting to return the signal handler to its previous state. Notably this occurred when using mod_python. The code was changed to use SIG_DFL when no old handler was defined.
- Corrected a bug in CompImageHDU that prevented rescaling the image data using hdu.scale(option='old').

2.1.1 (2009-04-22)

Updates described in this release are only supported in the NUMPY version of pyfits.

The following bugs were fixed:

- Corrected a bug that caused an exception to be raised when closing a file opened for append, where an HDU was appended to the file, after data was accessed from the file. This exception was only raised when running on a Windows platform.
- Updated the installation scripts, compression source code, and benchmark test scripts to properly install, build, and execute on a Windows platform.

2.1 (2009-04-14)

Updates described in this release are only supported in the NUMPY version of pyfits.

The following enhancements were made:

- Added new `tdump` and `tcreate` capabilities to `pyfits`.
 - The new `tdump` convenience function allows the contents of a binary table HDU to be dumped to a set of three files in ASCII format. One file will contain column definitions, the second will contain header parameters, and the third will contain header data.
 - The new `tcreate` convenience function allows the creation of a binary table HDU from the three files dumped by the `tdump` convenience function.
 - The primary use for the `tdump/tcreate` methods are to allow editing in a standard text editor of the binary table data and parameters.
- Added support for case sensitive values of the `EXTNAME` card in an extension header. (CNSHD745784)
 - By default, `pyfits` converts the value of `EXTNAME` cards to upper case when reading from a file. A new convenience function (`setExtensionNameCaseSensitive`) was implemented to allow a user to circumvent this behavior so that the `EXTNAME` value remains in the same case as it is in the file.
 - With the following function call, `pyfits` will maintain the case of all characters in the `EXTNAME` card values of all extension HDU's during the entire python session, or until another call to the function is made:

```
>>> import pyfits
>>> pyfits.setExtensionNameCaseSensitive()
```
 - The following function call will return `pyfits` to its default (all upper case) behavior:

```
>>> pyfits.setExtensionNameCaseSensitive(False)
```
- Added support for reading and writing FITS files in which the value of the first card in the header is 'SIMPLE=F'. In this case, the `pyfits` open function returns an `HDUList` object that contains a single HDU of the new type `_NonstandardHDU`. The header for this HDU is like a normal header (with the exception that the first card contains `SIMPLE=F` instead of `SIMPLE=T`). Like normal HDU's the reading of the data is delayed until actually requested. The data is read from the file into a string starting from the first byte after the header `END` card and continuing till the end of the file. When written, the header is written, followed by the data string. No attempt is made to pad the data string so that it fills into a standard 2880 byte FITS block. (CNSHD744730)
- Added support for FITS files containing extensions with unknown `XTENSION` card values. (CNSHD744730) Standard FITS files support extension HDU's of types `TABLE`, `IMAGE`, `BINTABLE`, and `A3DTABLE`. Accessing a nonstandard extension from a FITS file will now create a `_NonstandardExtHDU` object. Accessing the data of this object will cause the data to be read from the file into a string. If the HDU is written back to a file the string data is written after the Header and padded to fill a standard 2880 byte FITS block.

The following bugs were fixed:

- Extensive changes were made to the tiled image compression code to support the latest enhancements made in `CFITSIO` version 3.13 to support this convention.
- Eliminated a memory leak in the tiled image compression code.
- Corrected a bug in the `FITS_record.__setitem__` method which raised a `NameError` exception when attempting to set a value in a `FITS_record` object. (CNSHD745844)
- Corrected a bug that caused a `TypeError` exception to be raised when reading fits files containing large table HDU's (>2Gig). (CNSHD745522)
- Corrected a bug that caused a `TypeError` exception to be raised for all calls to the `warnings` module when running under Python 2.6. The `formatwarning` method in the `warnings` module was changed in Python 2.6 to include a new argument. (CNSHD746592)
- Corrected the behavior of the membership (`in`) operator in the `Header` class to check against header card keywords instead of card values. (CNSHD744730)

- Corrected the behavior of iteration on a Header object. The new behavior iterates over the unique card keywords instead of the card values.

2.0.1 (2009-02-03)

Updates described in this release are only supported in the NUMPY version of pyfits.

The following bugs were fixed:

- Eliminated a memory leak when reading Table HDU's from a fits file. (CNSHD741877)

2.0 (2009-01-30)

Updates described in this release are only supported in the NUMPY version of pyfits.

The following enhancements were made:

- Provide initial support for an image compression convention known as the “Tiled Image Compression Convention” [1].
 - The principle used in this convention is to first divide the n-dimensional image into a rectangular grid of subimages or “tiles”. Each tile is then compressed as a continuous block of data, and the resulting compressed byte stream is stored in a row of a variable length column in a FITS binary table. Several commonly used algorithms for compressing image tiles are supported. These include, GZIP, RICE, H-Compress and IRAF pixel list (PLIO).
 - Support for compressed image data is provided using the optional “pyfitsComp” module contained in a C shared library (pyfitsCompmodule.so).
 - The header of a compressed image HDU appears to the user like any image header. The actual header stored in the FITS file is that of a binary table HDU with a set of special keywords, defined by the convention, to describe the structure of the compressed image. The conversion between binary table HDU header and image HDU header is all performed behind the scenes. Since the HDU is actually a binary table, it may not appear as a primary HDU in a FITS file.
 - The data of a compressed image HDU appears to the user as standard uncompressed image data. The actual data is stored in the fits file as Binary Table data containing at least one column (COMPRESSED_DATA). Each row of this variable-length column contains the byte stream that was generated as a result of compressing the corresponding image tile. Several optional columns may also appear. These include, UNCOMPRESSED_DATA to hold the uncompressed pixel values for tiles that cannot be compressed, ZSCALE and ZZERO to hold the linear scale factor and zero point offset which may be needed to transform the raw uncompressed values back to the original image pixel values, and ZBLANK to hold the integer value used to represent undefined pixels (if any) in the image.
 - To create a compressed image HDU from scratch, simply construct a CompImageHDU object from an uncompressed image data array and its associated image header. From there, the HDU can be treated just like any image HDU:

```
>>> hdu=pyfits.CompImageHDU(imageData,imageHeader)
>>> hdu.writeto('compressed_image.fits')
```

- The signature for the CompImageHDU initializer method describes the possible options for constructing a CompImageHDU object:

```
def __init__(self, data=None, header=None, name=None,
             compressionType='RICE_1',
             tileSize=None,
             hcompScale=0.,
```

```

        hcompSmooth=0,
        quantizeLevel=16.):
    """
    data:          data of the image
    header:        header to be associated with the
                  image
    name:          the EXTNAME value; if this value
                  is None, then the name from the
                  input image header will be used;
                  if there is no name in the input
                  image header then the default name
                  'COMPRESSED_IMAGE' is used
    compressionType: compression algorithm 'RICE_1',
                  'PLIO_1', 'GZIP_1', 'HCOMPRESS_1'
    tileSize:      compression tile sizes default
                  treats each row of image as a tile
    hcompScale:    HCOMPRESS scale parameter
    hcompSmooth:   HCOMPRESS smooth parameter
    quantizeLevel: floating point quantization level;
    """

```

- Added two new convenience functions. The `setval` function allows the setting of the value of a single header card in a fits file. The `delval` function allows the deletion of a single header card in a fits file.
- A modification was made to allow the reading of data from a fits file containing a Table HDU that has duplicate field names. It is normally a requirement that the field names in a Table HDU be unique. Prior to this change a `ValueError` was raised, when the data was accessed, to indicate that the HDU contained duplicate field names. Now, a warning is issued and the field names are made unique in the internal record array. This will not change the `TTYPEn` header card values. You will be able to get the data from all fields using the field name, including the first field containing the name that is duplicated. To access the data of the other fields with the duplicated names you will need to use the field number instead of the field name. (CNSHD737193)
- An enhancement was made to allow the reading of unsigned integer 16 values from an ImageHDU when the data is signed integer 16 and `BZERO` is equal to 32784 and `BSCALE` is equal to 1 (the standard way for scaling unsigned integer 16 data). A new optional keyword argument (`uint16`) was added to the open convenience function. Supplying a value of `True` for this argument will cause data of this type to be read in and scaled into an unsigned integer 16 array, instead of a float 32 array. If a HDU associated with a file that was opened with the `uint16` option and containing unsigned integer 16 data is written to a file, the data will be reverse scaled into an integer 16 array and written out to the file and the `BSCALE/BZERO` header cards will be written with the values 1 and 32768 respectively. (CHSHD736064) Reference the following example:

```

>>> import pyfits
>>> hdul=pyfits.open('o4sp040b0_raw.fits',uint16=1)
>>> hdul[1].data
array([[1507, 1509, 1505, ..., 1498, 1500, 1487],
       [1508, 1507, 1509, ..., 1498, 1505, 1490],
       [1505, 1507, 1505, ..., 1499, 1504, 1491],
       ...,
       [1505, 1506, 1507, ..., 1497, 1502, 1487],
       [1507, 1507, 1504, ..., 1495, 1499, 1486],
       [1515, 1507, 1504, ..., 1492, 1498, 1487]], dtype=uint16)
>>> hdul.writeto('tmp.fits')
>>> hdul1=pyfits.open('tmp.fits',uint16=1)
>>> hdul1[1].data
array([[1507, 1509, 1505, ..., 1498, 1500, 1487],
       [1508, 1507, 1509, ..., 1498, 1505, 1490],
       [1505, 1507, 1505, ..., 1499, 1504, 1491],
       ...,

```

```

    [1505, 1506, 1507, ..., 1497, 1502, 1487],
    [1507, 1507, 1504, ..., 1495, 1499, 1486],
    [1515, 1507, 1504, ..., 1492, 1498, 1487]], dtype=uint16)
>>> hdull=pyfits.open('tmp.fits')
>>> hdull[1].data
array([[ 1507.,  1509.,  1505., ...,  1498.,  1500.,  1487.],
       [ 1508.,  1507.,  1509., ...,  1498.,  1505.,  1490.],
       [ 1505.,  1507.,  1505., ...,  1499.,  1504.,  1491.],
       ...,
       [ 1505.,  1506.,  1507., ...,  1497.,  1502.,  1487.],
       [ 1507.,  1507.,  1504., ...,  1495.,  1499.,  1486.],
       [ 1515.,  1507.,  1504., ...,  1492.,  1498.,  1487.]], dtype=float32)

```

- Enhanced the message generated when a ValueError exception is raised when attempting to access a header card with an unparseable value. The message now includes the Card name.

The following bugs were fixed:

- Corrected a bug that occurs when appending a binary table HDU to a fits file. Data was not being byteswapped on little endian machines. (CNSHD737243)
- Corrected a bug that occurs when trying to write an ImageHDU that is missing the required PCOUNT card in the header. An UnboundLocalError exception complaining that the local variable 'insert_pos' was referenced before assignment was being raised in the method _ValidHDU.req_cards. The code was modified so that it would properly issue a more meaningful ValueError exception with a description of what required card is missing in the header.
- Eliminated a redundant warning message about the PCOUNT card when validating an ImageHDU header with a PCOUNT card that is missing or has a value other than 0.

1.4.1 (2008-11-04)

Updates described in this release are only supported in the NUMPY version of pyfits.

The following enhancements were made:

- Enhanced the way import errors are reported to provide more information.

The following bugs were fixed:

- Corrected a bug that occurs when a card value is a string and contains a colon but is not a record-valued keyword card.
- Corrected a bug where pyfits fails to properly handle a record-valued keyword card with values using exponential notation and trailing blanks.

1.4 (2008-07-07)

Updates described in this release are only supported in the NUMPY version of pyfits.

The following enhancements were made:

- Added support for file objects and file like objects.
 - All convenience functions and class methods that take a file name will now also accept a file object or file like object. File like objects supported are StringIO and GzipFile objects. Other file like objects will work only if they implement all of the standard file object methods.

- For the most part, file or file like objects may be either opened or closed at function call. An opened object must be opened with the proper mode depending on the function or method called. Whenever possible, if the object is opened before the method is called, it will remain open after the call. This will not be possible when writing a HDUList that has been resized or when writing to a GzipFile object regardless of whether it is resized. If the object is closed at the time of the function call, only the name from the object is used, not the object itself. The pyfits code will extract the file name used by the object and use that to create an underlying file object on which the function will be performed.
- Added support for record-valued keyword cards as introduced in the “FITS WCS Paper IV proposal for representing a more general distortion model”.
 - Record-valued keyword cards are string-valued cards where the string is interpreted as a definition giving a record field name, and its floating point value. In a FITS header they have the following syntax:

```
keyword= 'field-specifier: float'
```

where keyword is a standard eight-character FITS keyword name, float is the standard FITS ASCII representation of a floating point number, and these are separated by a colon followed by a single blank.

The grammar for field-specifier is:

```
field-specifier:  
    field  
    field-specifier.field
```

```
field:  
    identifier  
    identifier.index
```

where identifier is a sequence of letters (upper or lower case), underscores, and digits of which the first character must not be a digit, and index is a sequence of digits. No blank characters may occur in the field-specifier. The index is provided primarily for defining array elements though it need not be used for that purpose.

Multiple record-valued keywords of the same name but differing values may be present in a FITS header. The field-specifier may be viewed as part of the keyword name.

Some examples follow:

```
DP1    = 'NAXIS: 2'  
DP1    = 'AXIS.1: 1'  
DP1    = 'AXIS.2: 2'  
DP1    = 'NAUX: 2'  
DP1    = 'AUX.1.COEFF.0: 0'  
DP1    = 'AUX.1.POWER.0: 1'  
DP1    = 'AUX.1.COEFF.1: 0.00048828125'  
DP1    = 'AUX.1.POWER.1: 1'
```

- As with standard header cards, the value of a record-valued keyword card can be accessed using either the index of the card in a HDU’s header or via the keyword name. When accessing using the keyword name, the user may specify just the card keyword or the card keyword followed by a period followed by the field-specifier. Note that while the card keyword is case insensitive, the field-specifier is not. Thus, `hdu['abc.def']`, `hdu['ABC.def']`, or `hdu['aBc.def']` are all equivalent but `hdu['ABC.DEF']` is not.
- When accessed using the card index of the HDU’s header the value returned will be the entire string value of the card. For example:

```
>>> print hdr[10]  
NAXIS: 2  
>>> print hdr[11]  
AXIS.1: 1
```

- When accessed using the keyword name exclusive of the field-specifier, the entire string value of the header card with the lowest index having that keyword name will be returned. For example:

```
>>> print hdr['DP1']
NAXIS: 2
```

- When accessing using the keyword name and the field-specifier, the value returned will be the floating point value associated with the record-valued keyword card. For example:

```
>>> print hdr['DP1.NAXIS']
2.0
```

- Any attempt to access a non-existent record-valued keyword card value will cause an exception to be raised (IndexError exception for index access or KeyError for keyword name access).

- Updating the value of a record-valued keyword card can also be accomplished using either index or keyword name. For example:

```
>>> print hdr['DP1.NAXIS']
2.0
>>> hdr['DP1.NAXIS'] = 3.0
>>> print hdr['DP1.NAXIS']
3.0
```

- Adding a new record-valued keyword card to an existing header is accomplished using the Header.update() method just like any other card. For example:

```
>>> hdr.update('DP1', 'AXIS.3: 1', 'a comment', after='DP1.AXIS.2')
```

- Deleting a record-valued keyword card from an existing header is accomplished using the standard list deletion syntax just like any other card. For example:

```
>>> del hdr['DP1.AXIS.1']
```

- In addition to accessing record-valued keyword cards individually using a card index or keyword name, cards can be accessed in groups using a set of special pattern matching keys. This access is made available via the standard list indexing operator providing a keyword name string that contains one or more of the special pattern matching keys. Instead of returning a value, a CardList object will be returned containing shared instances of the Cards in the header that match the given keyword specification.

- There are three special pattern matching keys. The first key '*' will match any string of zero or more characters within the current level of the field-specifier. The second key '?' will match a single character. The third key '...' must appear at the end of the keyword name string and will match all keywords that match the preceding pattern down all levels of the field-specifier. All combinations of ?, *, and ... are permitted (though ... is only permitted at the end). Some examples follow:

```
>>> cl=hdr['DP1.AXIS.*']
>>> print cl
DP1      = 'AXIS.1: 1'
DP1      = 'AXIS.2: 2'
>>> cl=hdr['DP1.*']
>>> print cl
DP1      = 'NAXIS: 2'
DP1      = 'NAUX: 2'
>>> cl=hdr['DP1.AUX...']
>>> print cl
DP1      = 'AUX.1.COEFF.0: 0'
DP1      = 'AUX.1.POWER.0: 1'
DP1      = 'AUX.1.COEFF.1: 0.00048828125'
DP1      = 'AUX.1.POWER.1: 1'
```

```
>>> cl=hdr['DP?.NAXIS']
>>> print cl
DP1      = 'NAXIS: 2'
DP2      = 'NAXIS: 2'
DP3      = 'NAXIS: 2'
>>> cl=hdr['DP1.A*S.*']
>>> print cl
DP1      = 'AXIS.1: 1'
DP1      = 'AXIS.2: 2'
```

- The use of the special pattern matching keys for adding or updating header cards in an existing header is not allowed. However, the deletion of cards from the header using the special keys is allowed. For example:

```
>>> del hdr['DP3.A*...']
```

- As noted above, accessing pyfits Header object using the special pattern matching keys will return a CardList object. This CardList object can itself be searched in order to further refine the list of Cards. For example:

```
>>> cl=hdr['DP1...']
>>> print cl
DP1      = 'NAXIS: 2'
DP1      = 'AXIS.1: 1'
DP1      = 'AXIS.2: 2'
DP1      = 'NAUX: 2'
DP1      = 'AUX.1.COEFF.1: 0.000488'
DP1      = 'AUX.2.COEFF.2: 0.00097656'
>>> c11=cl['*.AUX...']
>>> print c11
DP1      = 'NAUX: 2'
DP1      = 'AUX.1.COEFF.1: 0.000488'
DP1      = 'AUX.2.COEFF.2: 0.00097656'
```

- The CardList keys() method will allow the retrieval of all of the key values in the CardList. For example:

```
>>> cl=hdr['DP1.AXIS.*']
>>> print cl
DP1      = 'AXIS.1: 1'
DP1      = 'AXIS.2: 2'
>>> cl.keys()
['DP1.AXIS.1', 'DP1.AXIS.2']
```

- The CardList values() method will allow the retrieval of all of the values in the CardList. For example:

```
>>> cl=hdr['DP1.AXIS.*']
>>> print cl
DP1      = 'AXIS.1: 1'
DP1      = 'AXIS.2: 2'
>>> cl.values()
[1.0, 2.0]
```

- Individual cards can be retrieved from the list using standard list indexing. For example:

```
>>> cl=hdr['DP1.AXIS.*']
>>> c=c1[0]
>>> print c
DP1      = 'AXIS.1: 1'
>>> c=c1['DP1.AXIS.2']
>>> print c
DP1      = 'AXIS.2: 2'
```

- Individual card values can be retrieved from the list using the value attribute of the card. For example:

```
>>> cl=hdr['DP1.AXIS.*']
>>> cl[0].value
1.0
```

- The cards in the CardList are shared instances of the cards in the source header. Therefore, modifying a card in the CardList also modifies it in the source header. However, making an addition or a deletion to the CardList will not affect the source header. For example:

```
>>> hdr['DP1.AXIS.1']
1.0
>>> cl=hdr['DP1.AXIS.*']
>>> cl[0].value = 4.0
>>> hdr['DP1.AXIS.1']
4.0
>>> del cl[0]
>>> print cl['DP1.AXIS.1']
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "NP_pyfits.py", line 977, in __getitem__
    return self.ascard[key].value
File "NP_pyfits.py", line 1258, in __getitem__
    _key = self.index_of(key)
File "NP_pyfits.py", line 1403, in index_of
    raise KeyError, 'Keyword %s not found.' % `key`
KeyError: "Keyword 'DP1.AXIS.1' not found."
>>> hdr['DP1.AXIS.1']
4.0
```

- A FITS header consists of card images. In pyfits each card image is manifested by a Card object. A pyfits Header object contains a list of Card objects in the form of a CardList object. A record-valued keyword card image is represented in pyfits by a RecordValuedKeywordCard object. This object inherits from a Card object and has all of the methods and attributes of a Card object.
- A new RecordValuedKeywordCard object is created with the RecordValuedKeywordCard constructor: RecordValuedKeywordCard(key, value, comment). The key and value arguments may be specified in two ways. The key value may be given as the 8 character keyword only, in which case the value must be a character string containing the field-specifier, a colon followed by a space, followed by the actual value. The second option is to provide the key as a string containing the keyword and field-specifier, in which case the value must be the actual floating point value. For example:

```
>>> c1 = pyfits.RecordValuedKeywordCard('DP1', 'NAXIS: 2', 'Number of variables')
>>> c2 = pyfits.RecordValuedKeywordCard('DP1.AXIS.1', 1.0, 'Axis number')
```

- RecordValuedKeywordCards have attributes .key, .field_specifier, .value, and .comment. Both .value and .comment can be changed but not .key or .field_specifier. The constructor will extract the field-specifier from the input key or value, whichever is appropriate. The .key attribute is the 8 character keyword.
- Just like standard Cards, a RecordValuedKeywordCard may be constructed from a string using the fromstring() method or verified using the verify() method. For example:

```
>>> c1 = pyfits.RecordValuedKeywordCard().fromstring(
    "DP1      = 'NAXIS: 2' / Number of independent variables")
>>> c2 = pyfits.RecordValuedKeywordCard().fromstring(
    "DP1      = 'AXIS.1: X' / Axis number")
>>> print c1; print c2
DP1      = 'NAXIS: 2' / Number of independent variables
DP1      = 'AXIS.1: X' / Axis number
>>> c2.verify()
```

```
Output verification result:
Card image is not FITS standard (unparsable value string).
```

- A standard card that meets the criteria of a `RecordValuedKeywordCard` may be turned into a `RecordValuedKeywordCard` using the class method `coerce`. If the card object does not meet the required criteria then the original card object is just returned.

```
>>> c1 = pyfits.Card('DP1', 'AUX: 1', 'comment')
>>> c2 = pyfits.RecordValuedKeywordCard.coerce(c1)
>>> print type(c2)
<'pyfits.NP_pyfits.RecordValuedKeywordCard'>
```

- Two other card creation methods are also available as `RecordValuedKeywordCard` class methods. These are `createCard()` which will create the appropriate card object (`Card` or `RecordValuedKeywordCard`) given input key, value, and comment, and `createCardFromString` which will create the appropriate card object given an input string. These two methods are also available as convenience functions:

```
>>> c1 = pyfits.RecordValuedKeywordCard.createCard('DP1', 'AUX: 1', 'comment')
```

or

```
>>> c1 = pyfits.createCard('DP1', 'AUX: 1', 'comment')
>>> print type(c1)
<'pyfits.NP_pyfits.RecordValuedKeywordCard'>
```

```
>>> c1 = pyfits.RecordValuedKeywordCard.createCard('DP1', 'AUX 1', 'comment')
```

or

```
>>> c1 = pyfits.createCard('DP1', 'AUX 1', 'comment')
>>> print type(c1)
<'pyfits.NP_pyfits.Card'>
```

```
>>> c1 = pyfits.RecordValuedKeywordCard.createCardFromString \
      ("DP1 = 'AUX: 1.0' / comment")
```

or

```
>>> c1 = pyfits.createCardFromString("DP1      = 'AUX: 1.0' / comment")
>>> print type(c1)
<'pyfits.NP_pyfits.RecordValuedKeywordCard'>
```

The following bugs were fixed:

- Corrected a bug that occurs when writing a HDU out to a file. During the write, any Keyboard Interrupts are trapped so that the write completes before the interrupt is handled. Unfortunately, the Keyboard Interrupt was not properly reinstated after the write completed. This was fixed. (CNSHD711138)
- Corrected a bug when using `ipython`, where temporary files created with the `tempFile.NamedTemporaryFile` method are not automatically removed. This can happen for instance when opening a Gzipped fits file or when open a fits file over the internet. The files will now be removed. (CNSHD718307)
- Corrected a bug in the `append` convenience function's call to the `writeto` convenience function. The `classExtensions` argument must be passed as a keyword argument.
- Corrected a bug that occurs when retrieving variable length character arrays from binary table HDUs (PA() format) and using slicing to obtain rows of data containing variable length arrays. The code issued a `TypeError` exception. The data can now be accessed with no exceptions. (CNSHD718749)

- Corrected a bug that occurs when retrieving data from a fits file opened in memory map mode when the file contains multiple image extensions or ASCII table or binary table HDUs. The code issued a `TypeError` exception. The data can now be accessed with no exceptions. (CNSHD707426)
- Corrected a bug that occurs when attempting to get a subset of data from a Binary Table HDU and then use the data to create a new Binary Table HDU object. A `TypeError` exception was raised. The data can now be subsetted and used to create a new HDU. (CNSHD723761)
- Corrected a bug that occurs when attempting to scale an Image HDU back to its original data type using the `_ImageBaseHDU.scale` method. The code was not resetting the BITPIX header card back to the original data type. This has been corrected.
- Changed the code to issue a `KeyError` exception instead of a `NameError` exception when accessing a non-existent field in a table.

1.3 (2008-02-22)

Updates described in this release are only supported in the NUMPY version of pyfits.

The following enhancements were made:

- Provided support for a new extension to pyfits called *stpyfits*.
 - The *stpyfits* module is a wrapper around pyfits. It provides all of the features and functions of pyfits along with some STScI specific features. Currently, the only new feature supported by stpyfits is the ability to read and write fits files that contain image data quality extensions with constant data value arrays. See stpyfits [2] for more details on stpyfits.
- Added a new feature to allow trailing HDUs to be deleted from a fits file without actually reading the data from the file.
 - This supports a JWST requirement to delete a trailing HDU from a file whose primary Image HDU is too large to be read on a 32 bit machine.
- Updated pyfits to use the warnings module to issue warnings. All warnings will still be issued to stdout, exactly as they were before, however, you may now suppress warnings with the `-Wignore` command line option. For example, to run a script that will ignore warnings use the following command line syntax:


```
python -Wignore yourscrip.py
```
- Updated the open convenience function to allow the input of an already opened file object in place of a file name when opening a fits file.
- Updated the writeto convenience function to allow it to accept the `output_verify` option.
 - In this way, the user can use the argument `output_verify='fix'` to allow pyfits to correct any errors it encounters in the provided header before writing the data to the file.
- Updated the verification code to provide additional detail with a `VerifyError` exception.
- Added the capability to create a binary table HDU directly from a `numpy.ndarray`. This may be done using either the `new_table` convenience function or the `BinTableHDU` constructor.

The following performance improvements were made:

- Modified the import logic to dramatically decrease the time it takes to import pyfits.
- Modified the code to provide performance improvements when copying and examining header cards.

The following bugs were fixed:

- Corrected a bug that occurs when reading the data from a fits file that includes BZERO/BSCALE scaling. When the data is read in from the file, pyfits automatically scales the data using the BZERO/BSCALE values in the

header. In the previous release, pyfits created a 32 bit floating point array to hold the scaled data. This could cause a problem when the value of BZERO is so large that the scaled value will not fit into the float 32. For this release, when the input data is 32 bit integer, a 64 bit floating point array is used for the scaled data.

- Corrected a bug that caused an exception to be raised when attempting to scale image data using the ImageHDU.scale method.
- Corrected a bug in the new_table convenience function that occurred when a binary table was created using a ColDefs object as input and supplying an nrows argument for a number of rows that is greater than the number of rows present in the input ColDefs object. The previous version of pyfits failed to allocate the necessary memory for the additional rows.
- Corrected a bug in the new_table convenience function that caused an exception to be thrown when creating an ASCII table.
- Corrected a bug in the new_table convenience function that will allow the input of a ColDefs object that was read from a file as a binary table with a data value equal to None.
- Corrected a bug in the construction of ASCII tables from Column objects that are created with noncontinuous start columns.
- Corrected bugs in a number of areas that would sometimes cause a failure to improperly raise an exception when an error occurred.
- Corrected a bug where attempting to open a non-existent fits file on a windows platform using a drive letter in the file specification caused a misleading IOError exception to be raised.

1.1 (2007-06-15)

- Modified to use either NUMPY or NUMARRAY.
- New file writing modes have been provided to allow streaming data to extensions without requiring the whole output extension image in memory. See documentation on StreamingHDU.
- Improvements to minimize byteswapping and memory usage by byteswapping in place.
- Now supports ':' characters in filenames.
- Handles keyboard interrupts during long operations.
- Preserves the byte order of the input image arrays.

1.0.1 (2006-03-24)

The changes to PyFITS were primarily to improve the docstrings and to reclassify some public functions and variables as private. Readgeis and fitsdiff which were distributed with PyFITS in previous releases were moved to pytools. This release of PyFITS is v1.0.1. The next release of PyFITS will support both numarray and numpy (and will be available separately from stsci_python, as are all the python packages contained within stsci_python). An alpha release for PyFITS numpy support will be made around the time of this stsci_python release.

- Updated docstrings for public functions.
- Made some previously public functions private.

1.0 (2005-11-01)

Major Changes since v0.9.6:

- Added support for the HIERARCH convention

- Added support for iteration and slicing for HDU lists
- PyFITS now uses the standard setup.py installation script
- Add utility functions at the module level, they include:
 - getheader
 - getdata
 - getval
 - writeto
 - append
 - update
 - info

Minor changes since v0.9.6:

- Fix a bug to make single-column ASCII table work.
- Fix a bug so a new table constructed from an existing table with X-formatted columns will work.
- Fix a problem in verifying HDUList right after the open statement.
- Verify that elements in an HDUList, besides the first one, are ExtensionHDU.
- Add output verification in methods flush() and close().
- Modify the the design of the open() function to remove the output_verify argument.
- Remove the groups argument in GroupsHDU's constructor.
- Redesign the column definition class to make its column components more accessible. Also to make it conducive for higher level functionalities, e.g. combining two column definitions.
- Replace the Boolean class with the Python Boolean type. The old TRUE/FALSE will still work.
- Convert classes to the new style.
- Better format when printing card or card list.
- Add the optional argument clobber to all writeto() functions and methods.
- If adding a blank card, will not use existing blank card's space.

PyFITS Version 1.0 REQUIRES Python 2.3 or later.

0.9.6 (2004-11-11)

Major changes since v0.9.3:

- Support for variable length array tables.
- Support for writing ASCII table extensions.
- Support for random groups, both reading and writing.

Some minor changes:

- Support for numbers with leading zeros in an ASCII table extension.
- Changed scaled columns' data type from Float32 to Float64 to preserve precision.
- Made Column constructor more flexible in accepting format specification.

0.9.3 (2004-07-02)

Changes since v0.9.0:

- Lazy instantiation of full Headers/Cards for all HDU's when the file is opened. At the open, only extracts vital info (e.g. NAXIS's) from the header parts. This change will speed up the performance if the user only needs to access one extension in a multi-extension FITS file.
- Support the X format (bit flags) columns, both reading and writing, in a binary table. At the user interface, they are converted to Boolean arrays for easy manipulation. For example, if the column's TFORM is "11X", internally the data is stored in 2 bytes, but the user will see, at each row of this column, a Boolean array of 11 elements.
- Fix a bug such that when a table extension has no data, it will not try to scale the data when updating/writing the HDU list.

0.9 (2004-04-27)

Changes since v0.8.0:

- Rewriting of the Card class to separate the parsing and verification of header cards
- Restructure the keyword indexing scheme which speed up certain applications (update large number of new keywords and reading a header with larger numbers of cards) by a factor of 30 or more
- Change the default to be lenient FITS standard checking on input and strict FITS standard checking on output
- Support CONTINUE cards, both reading and writing
- Verification can now be performed at any of the HDUList, HDU, and Card levels
- Support (contiguous) subsection (attribute .section) of images to reduce memory usage for large images

0.8.0 (2003-08-19)

NOTE: This version will only work with numarray Version 0.6. In addition, earlier versions of PyFITS will not work with numarray 0.6. Therefore, both must be updated simultaneously.

Changes since 0.7.6:

- Compatible with numarray 0.6/records 2.0
- For binary tables, now it is possible to update the original array if a scaled field is updated.
- Support of complex columns
- Modify the `__getitem__` method in `FITS_rec`. In order to make sure the scaled quantities are also viewing the same data as the original `FITS_rec`, all fields need to be "touched" when `__getitem__` is called.
- Add a new attribute `mmap` for `HDUList`, and close the `mmap` object when close `HDUList` object. Earlier version does not close `mmap` object and can cause memory lockup.
- Enable 'update' as a legitimate `mmap` mode.
- Do not print message when closing an `HDUList` object which is not created from reading a FITS file. Such message is confusing.
- remove the internal attribute "closed" and related method (`__getattr__` in `HDUList`). It is redundant.

0.7.6 (2002-11-22)

NOTE: This version will only work with numarray Version 0.4.

Changes since 0.7.5:

- Change `x*=n` to `numarray.multiply(x, n, x)` where `n` is a floating number, in order to make `pyfits` to work under Python 2.2. (2 occurrences)
- Modify the “update” method in the Header class to use the “fixed-format” card even if the card already exists. This is to avoid the mis-alignment as shown below:

After running `drizzle` on ACS images it creates a CD matrix whose elements have very many digits, *e.g.*:

```
CD1_1 = 1.1187596304411E-05 / partial of first axis coordinate w.r.t. x CD1_2 = -
8.502767249350019E-06 / partial of first axis coordinate w.r.t. y
```

with `pyfits`, an “update” on these header items and write in new values which has fewer digits, *e.g.*:

```
CD1_1 = 1.0963011E-05 / partial of first axis coordinate w.r.t. x CD1_2 = -8.527229E-06 / partial of
first axis coordinate w.r.t. y
```

- Change some internal variables to make their appearance more consistent:

old name new name

```
__octalRegex __octalRegex __readblock() _readblock() __formatter() _formatter(). __value_RE
_value_RE __numr __numr __comment_RE _comment_RE __keywd_RE _keywd_RE __num-
ber_RE _number_RE. tmpName() _tmpName() dimShape _dimShape ErrList _ErrList
```

- Move up the module description. Move the copyright statement to the bottom and assign to the variable `__credits__`.
- change the following line:

```
self.__dict__ = input.__dict__
```

to

```
self.__setstate__(input.__getstate__())
```

in order for `pyfits` to run under `numarray 0.4`.

- edit `_readblock` to add the (optional) `firstblock` argument and raise `IOError` if the the first 8 characters in the first block is not ‘SIMPLE ‘ or ‘XTENSION’. Edit the function open to check for `IOError` to skip the last null filled block(s). Edit `readHDU` to add the `firstblock` argument.

0.7.5 (2002-08-16)

Changes since v0.7.3:

- Memory mapping now works for readonly mode, both for images and binary tables.
Usage: `pyfits.open('filename', memmap=1)`
- Edit the field method in `FITS_rec` class to make the column scaling for numbers use less temporary memory. (does not work under 2.2, due to Python “bug” of array `*=`)
- Delete `b scale/bzero` in the `ImageBaseHDU` constructor.
- Update `bitpix` in `BaseImageHDU.__getattr__` after deleting `b scale/bzero`. (bug fix)
- In `BaseImageHDU.__getattr__` point `self.data` to `raw_data` if float and if not `memmap`. (bug fix).
- Change the function `get_tbdata()` to private: `_get_tbdata()`.

0.7.3 (2002-07-12)

Changes since v0.7.2:

- It will scale all integer image data to Float32, if BSCALE/BZERO != 1/0. It will also expunge the BSCALE/BZERO keywords.
- Add the `scale()` method for ImageBaseHDU, so data can be scaled just before being written to the file. It has the following arguments:
 - type: destination data type (string), e.g. Int32, Float32, UInt8, etc.
 - option: scaling scheme. if 'old', use the old BSCALE/BZERO values. if 'minmax', use the data range to fit into the full range of specified integer type. Float destination data type will not be scaled for this option.
 - bSCALE/bzero: user specifiable BSCALE/BZERO values. They overwrite the "option".
- Deal with data area resizing in 'update' mode.
- Make the data scaling (both input and output) faster and use less memory.
- Bug fix to make column name change takes effect for field.
- Bug fix to avoid exception if the key is not present in the header already. This affects (fixes) `add_history()`, `add_comment()`, and `add_blank()`.
- Bug fix in `__getattr__()` in Card class. The change made in 0.7.2 to `rstrip` the comment must be string type to avoid exception.

0.7.2.1 (2002-06-25)

A couple of bugs were addressed in this version.

- Fix a bug in `_add_commentary()`. Due to a change in `index_of()` during version 0.6.5.5, `_add_commentary` needs to be modified to avoid exception if the key is not present in the header already. This affects (fixes) `add_history()`, `add_comment()`, and `add_blank()`.
- Fix a bug in `__getattr__()` in Card class. The change made in 0.7.2 to `rstrip` the comment must be string type to avoid exception.

0.7.2 (2002-06-19)

The two major improvements from Version 0.6.2 are:

- support reading tables with "scaled" columns (e.g. `tscal/tzero`, Boolean, and ASCII tables)
- a prototype output verification.

This version of PyFITS requires `numarray` version 0.3.4.

Other changes include:

- Implement the new HDU hierarchy proposed earlier this year. This in turn reduces some of the redundant methods common to several HDU classes.
- Add 3 new methods to the Header class: `add_history`, `add_comment`, and `add_blank`.
- The table attributes `_columns` are now `.columns` and the attributes in `ColDefs` are now all without the under-scores. So, a user can get a list of column names by: `hdu.columns.names`.
- The "fill" argument in the `new_table` method now has a new meaning:
If set to true (=1), it will fill the entire new table with zeros/blanks. Otherwise (=0), just the extra rows/cells are filled with zeros/blanks. Fill values other than zero/blank are now not possible.

- Add the argument `output_verify` to the `open` method and `writeto` method. Not in the `flush` or `close` methods yet, due to possible complication.
- A new `copy` method for tables, the copy is totally independent from the table it copies from.
- The `tostring()` call in `writeHDUdata` takes up extra space to store the string object. Use `tofile()` instead, to save space.
- Make changes from `_byteswap` to `_byteorder`, following corresponding changes in `numarray` and `recarray`.
- Insert(`update`) `EXTEND` in `PrimaryHDU` only when header is `None`.
- Strip the trailing blanks for the comment value of a card.
- Add `seek(0)` right after the `__buildin__.open(0)`, because for the 'ab+' mode, the pointer is at the end after open in Linux, but it is at the beginning in Solaris.
- Add checking of data against header, update header keywords (NAXIS's, BITPIX) when they don't agree with the data.
- change version to `__version__`.

There are also many other minor internal bug fixes and technical changes.

0.6.2 (2002-02-12)

This version requires `numarray` version 0.2.

Things not yet supported but are part of future development:

- Verification and/or correction of FITS objects being written to disk so that they are legal FITS. This is being added now and should be available in about a month. Currently, one may construct FITS headers that are inconsistent with the data and write such FITS objects to disk. Future versions will provide options to either a) correct discrepancies and warn, b) correct discrepancies silently, c) throw a Python exception, or d) write illegal FITS (for test purposes!).
- Support for `ascii` tables or random groups format. Support for ASCII tables will be done soon (~1 month). When random group support is added is uncertain.
- Support for memory mapping FITS data (to reduce memory demands). We expect to provide this capability in about 3 months.
- Support for columns in binary tables having scaled values (e.g. `BSCALE` or `BZERO`) or boolean values. Currently booleans are stored as `Int8` arrays and users must explicitly convert them into a boolean array. Likewise, scaled columns must be copied with scaling and offset by testing for those attributes explicitly. Future versions will produce such copies automatically.
- Support for tables with `TNULL` values. This awaits an enhancement to `numarray` to support mask arrays (planned). (At least a couple of months off).

14.5 Reference/API

A package for reading and writing FITS files and manipulating their contents.

A module for reading and writing Flexible Image Transport System (FITS) files. This file format was endorsed by the International Astronomical Union in 1999 and mandated by NASA as the standard format for storing high energy astrophysics data. For details of the FITS standard, see the NASA/Science Office of Standards and Technology publication, NOST 100-2.0.

14.5.1 File Handling and Convenience Functions

`open()`

`astropy.io.fits.open` (*name*, *mode*='readonly', *memmap*=None, *save_backup*=False, ***kwargs*)
Factory function to open a FITS file and return an `HDUList` object.

Parameters

name : file path, file object or file-like object

File to be opened.

mode : str

Open mode, 'readonly' (default), 'update', 'append', 'denywrite', or 'ostream'.

If *name* is a file object that is already opened, *mode* must match the mode the file was opened with, `readonly` (`rb`), `update` (`rb+`), `append` (`ab+`), `ostream` (`w`), `denywrite` (`rb`)).

memmap : bool

Is memory mapping to be used?

save_backup : bool

If the file was opened in `update` or `append` mode, this ensures that a backup of the original file is saved before any changes are flushed. The backup has the same name as the original file with ".bak" appended. If "file.bak" already exists then "file.bak.1" is used, and so on.

kwargs : dict

optional keyword arguments, possible values are:

• **uint** : bool

Interpret signed integer data where `BZERO` is the central value and `BSCALE == 1` as unsigned integer data. For example, `int16` data with `BZERO = 32768` and `BSCALE = 1` would be treated as `uint16` data.

Note, for backward compatibility, the kwarg **uint16** may be used instead. The kwarg was renamed when support was added for integers of any size.

• **ignore_missing_end** : bool

Do not issue an exception when opening a file that is missing an `END` card in the last header.

• **checksum** : bool, str

If `True`, verifies that both `DATASUM` and `CHECKSUM` card values (when present in the HDU header) match the header and data of all HDU's in the file. Updates to a file that already has a checksum will preserve and update the existing checksums unless this argument is given a value of 'remove', in which case the `CHECKSUM` and `DATASUM` values are not checked, and are removed when saving changes to the file.

• **disable_image_compression** : bool

If `True`, treats compressed image HDU's like normal binary table HDU's.

• **do_not_scale_image_data** : bool

If `True`, image data is not scaled using `BSCALE/BZERO` values when read.

•ignore_blank

[bool] If `True`, the BLANK keyword is ignored if present.

•scale_back : bool

If `True`, when saving changes to a file that contained scaled image data, restore the data to the original type and reapply the original BSCALE/BZERO values. This could lead to loss of accuracy if scaling back to integer values after performing floating point operations on the data.

Returns

hdulist : an `HDUList` object

`HDUList` containing all of the header data units in the file.

writeto()

`astropy.io.fits.writeto(filename, data, header=None, output_verify='exception', clobber=False, checksum=False)`

Create a new FITS file using the supplied data/header.

Parameters

filename : file path, file object, or file like object

File to write to. If opened, must be opened in a writeable binary mode such as 'wb' or 'ab+'.

data : array, record array, or groups data object

data to write to the new file

header : `Header` object, optional

the header associated with `data`. If `None`, a header of the appropriate type is created for the supplied data. This argument is optional.

output_verify : str

Output verification option. Must be one of "fix", "silentfix", "ignore", "warn", or "exception". May also be any combination of "fix" or "silentfix" with "+ignore", "+warn, or +exception" (e.g. ``"fix+warn"). See [Verification options](#) for more info.

clobber : bool, optional

If `True`, and if `filename` already exists, it will overwrite the file. Default is `False`.

checksum : bool, optional

If `True`, adds both DATASUM and CHECKSUM cards to the headers of all HDU's written to the file.

info()

`astropy.io.fits.info(filename, output=None, **kwargs)`

Print the summary information on a FITS file.

This includes the name, type, length of header, data shape and type for each extension.

Parameters

filename : file path, file object, or file like object

FITS file to obtain info from. If opened, mode must be one of the following: `rb`, `rb+`, or `ab+` (i.e. the file must be readable).

output : file, bool, optional

A file-like object to write the output to. If `False`, does not output to a file and instead returns a list of tuples representing the HDU info. Writes to `sys.stdout` by default.

kwargs

Any additional keyword arguments to be passed to `astropy.io.fits.open`. *Note:* This function sets `ignore_missing_end=True` by default.

`append()`

`astropy.io.fits.append(filename, data, header=None, checksum=False, verify=True, **kwargs)`

Append the header/data to FITS file if filename exists, create if not.

If only data is supplied, a minimal header is created.

Parameters

filename : file path, file object, or file like object

File to write to. If opened, must be opened for update (`rb+`) unless it is a new file, then it must be opened for append (`ab+`). A file or `GzipFile` object opened for update will be closed after return.

data : array, table, or group data object

the new data used for appending

header : `Header` object, optional

The header associated with `data`. If `None`, an appropriate header will be created for the data object supplied.

checksum : bool, optional

When `True` adds both `DATASUM` and `CHECKSUM` cards to the header of the HDU when written to the file.

verify : bool, optional

When `True`, the existing FITS file will be read in to verify it for correctness before appending. When `False`, content is simply appended to the end of the file. Setting `verify` to `False` can be much faster.

kwargs

Any additional keyword arguments to be passed to `astropy.io.fits.open`.

`update()`

`astropy.io.fits.update(filename, data, *args, **kwargs)`

Update the specified extension with the input data/header.

Parameters

filename : file path, file object, or file like object

File to update. If opened, mode must be update (`rb+`). An opened file object or `GzipFile` object will be closed upon return.

data : array, table, or group data object

the new data used for updating

header : `Header` object, optional

The header associated with `data`. If `None`, an appropriate header will be created for the data object supplied.

ext, extname, extver

The rest of the arguments are flexible: the 3rd argument can be the header associated with the data. If the 3rd argument is not a `Header`, it (and other positional arguments) are assumed to be the extension specification(s). Header and extension specs can also be keyword arguments. For example:

```
update(file, dat, hdr, 'sci') # update the 'sci' extension
update(file, dat, 3) # update the 3rd extension
update(file, dat, hdr, 3) # update the 3rd extension
update(file, dat, 'sci', 2) # update the 2nd SCI extension
update(file, dat, 3, header=hdr) # update the 3rd extension
update(file, dat, header=hdr, ext=5) # update the 5th extension
```

kwargs

Any additional keyword arguments to be passed to `astropy.io.fits.open`.

getdata ()

`astropy.io.fits.getdata (filename, *args, **kwargs)`

Get the data from an extension of a FITS file (and optionally the header).

Parameters

filename : file path, file object, or file like object

File to get data from. If opened, mode must be one of the following `rb`, `rb+`, or `ab+`.

ext

The rest of the arguments are for extension specification. They are flexible and are best illustrated by examples.

No extra arguments implies the primary header:

```
getdata('in.fits')
```

By extension number:

```
getdata('in.fits', 0) # the primary header
getdata('in.fits', 2) # the second extension
getdata('in.fits', ext=2) # the second extension
```

By name, i.e., `EXTNAME` value (if unique):

```
getdata('in.fits', 'sci')
getdata('in.fits', extname='sci') # equivalent
```

Note `EXTNAME` values are not case sensitive

By combination of `EXTNAME` and `EXTVER` as separate arguments or as a tuple:

```
getdata('in.fits', 'sci', 2) # EXTNAME='SCI' & EXTVER=2
getdata('in.fits', extname='sci', extver=2) # equivalent
getdata('in.fits', ('sci', 2)) # equivalent
```

Ambiguous or conflicting specifications will raise an exception:

```
getdata('in.fits', ext=('sci',1), extname='err', extver=2)
```

header : bool, optional

If `True`, return the data and the header of the specified HDU as a tuple.

lower, upper : bool, optional

If `lower` or `upper` are `True`, the field names in the returned data object will be converted to lower or upper case, respectively.

view : ndarray, optional

When given, the data will be returned wrapped in the given ndarray subclass by calling:

```
data.view(view)
```

kwargs

Any additional keyword arguments to be passed to `astropy.io.fits.open`.

Returns

array : array, record array or groups data object

Type depends on the type of the extension being referenced.

If the optional keyword `header` is set to `True`, this function will return a (data, header) tuple.

getheader ()

```
astropy.io.fits.getheader (filename, *args, **kwargs)
```

Get the header from an extension of a FITS file.

Parameters

filename : file path, file object, or file like object

File to get header from. If an opened file object, its mode must be one of the following `rb`, `rb+`, or `ab+`).

ext, extname, extver

The rest of the arguments are for extension specification. See the `getdata` documentation for explanations/examples.

kwargs

Any additional keyword arguments to be passed to `astropy.io.fits.open`.

Returns

header : `Header` object

getval()

`astropy.io.fits.getval(filename, keyword, *args, **kwargs)`

Get a keyword's value from a header in a FITS file.

Parameters

filename : file path, file object, or file like object

Name of the FITS file, or file object (if opened, mode must be one of the following `rb`, `rb+`, or `ab+`).

keyword : str

Keyword name

ext, extname, extver

The rest of the arguments are for extension specification. See `getdata` for explanations/examples.

kwargs

Any additional keyword arguments to be passed to `astropy.io.fits.open`.

Note: This function automatically specifies `do_not_scale_image_data = True` when opening the file so that values can be retrieved from the unmodified header.

Returns

keyword value : str, int, or float

setval()

`astropy.io.fits.setval(filename, keyword, *args, **kwargs)`

Set a keyword's value from a header in a FITS file.

If the keyword already exists, its value/comment will be updated. If it does not exist, a new card will be created and it will be placed before or after the specified location. If no `before` or `after` is specified, it will be appended at the end.

When updating more than one keyword in a file, this convenience function is a much less efficient approach compared with opening the file for update, modifying the header, and closing the file.

Parameters

filename : file path, file object, or file like object

Name of the FITS file, or file object If opened, mode must be update (`rb+`). An opened file object or `GzipFile` object will be closed upon return.

keyword : str

Keyword name

value : str, int, float, optional

Keyword value (default: `None`, meaning don't modify)

comment : str, optional

Keyword comment, (default: `None`, meaning don't modify)

before : str, int, optional

Name of the keyword, or index of the card before which the new card will be placed. The argument `before` takes precedence over `after` if both are specified (default: `None`).

after : str, int, optional

Name of the keyword, or index of the card after which the new card will be placed. (default: `None`).

savecomment : bool, optional

When `True`, preserve the current comment for an existing keyword. The argument `savecomment` takes precedence over `comment` if both specified. If `comment` is not specified then the current comment will automatically be preserved (default: `False`).

ext, extname, extver

The rest of the arguments are for extension specification. See `getdata` for explanations/examples.

kwargs

Any additional keyword arguments to be passed to `astropy.io.fits.open`. *Note:* This function automatically specifies `do_not_scale_image_data = True` when opening the file so that values can be retrieved from the unmodified header.

`delval()`

`astropy.io.fits.delval` (*filename, keyword, *args, **kwargs*)

Delete all instances of keyword from a header in a FITS file.

Parameters

filename : file path, file object, or file like object

Name of the FITS file, or file object. If opened, mode must be update (`rb+`). An opened file object or `GzipFile` object will be closed upon return.

keyword : str, int

Keyword name or index

ext, extname, extver

The rest of the arguments are for extension specification. See `getdata` for explanations/examples.

kwargs

Any additional keyword arguments to be passed to `astropy.io.fits.open`. *Note:* This function automatically specifies `do_not_scale_image_data = True` when opening the file so that values can be retrieved from the unmodified header.

14.5.2 HDU Lists

`HDUList`

class `astropy.io.fits.HDUList` (*hdus=[], file=None*)

Bases: `list`, `astropy.io.fits.verify._Verify`

HDU list class. This is the top-level FITS object. When a FITS file is opened, a `HDUList` object is returned.

Construct a `HDUList` object.

Parameters

hdus : sequence of HDU objects or single HDU, optional

The HDU object(s) to comprise the `HDUList`. Should be instances of HDU classes like `ImageHDU` or `BinTableHDU`.

file : file object, optional

The opened physical file associated with the `HDUList`.

append (*hdu*)

Append a new HDU to the `HDUList`.

Parameters

hdu : HDU object

HDU to add to the `HDUList`.

close (*output_verify='exception', verbose=False, closed=True*)

Close the associated FITS file and memmap object, if any.

Parameters

output_verify : str

Output verification option. Must be one of "fix", "silentfix", "ignore", "warn", or "exception". May also be any combination of "fix" or "silentfix" with "+ignore", "+warn, or +exception" (e.g. ``"fix+warn"). See [Verification options](#) for more info.

verbose : bool

When `True`, print out verbose messages.

closed : bool

When `True`, close the underlying file object.

fileinfo (*index*)

Returns a dictionary detailing information about the locations of the indexed HDU within any associated file. The values are only valid after a read or write of the associated file with no intervening changes to the `HDUList`.

Parameters

index : int

Index of HDU for which info is to be returned.

Returns

fileinfo : dict or None

The dictionary details information about the locations of the indexed HDU within an associated file. Returns `None` when the HDU is not associated with a file.

Dictionary contents:

Key	Value
file	File object associated with the HDU
file-name	Name of associated file object
file-mode	Mode in which the file was opened (readonly, update, append, denywrite, ostream)
re-sized	Flag that when <code>True</code> indicates that the data has been resized since the last read/write so the returned values may not be valid.
hdr-loc	Starting byte location of header in file
data-loc	Starting byte location of data block in file
dataSpan	Data size including padding

filename ()

Return the file name associated with the HDUList object if one exists. Otherwise returns None.

Returns

filename : a string containing the file name associated with the HDUList object if an association exists. Otherwise returns None.

flush (*args, **kwargs)

Force a write of the HDUList back to the file (for append and update modes only).

Parameters

output_verify : str

Output verification option. Must be one of "fix", "silentfix", "ignore", "warn", or "exception". May also be any combination of "fix" or "silentfix" with "+ignore", "+warn, or "+exception" (e.g. ``"fix+warn"). See *Verification options* for more info.

verbose : bool

When `True`, print verbose messages

classmethod fromfile (fileobj, mode=None, memmap=False, save_backup=False, **kwargs)

Creates an HDUList instance from a file-like object.

The actual implementation of `fitsopen()`, and generally shouldn't be used directly. Use `open()` instead (and see its documentation for details of the parameters accepted by this method).

classmethod fromstring (data, **kwargs)

Creates an HDUList instance from a string or other in-memory data buffer containing an entire FITS file. Similar to `HDUList.fromfile()`, but does not accept the mode or memmap arguments, as they are only relevant to reading from a file on disk.

This is useful for interfacing with other libraries such as CFITSIO, and may also be useful for streaming applications.

Parameters

data : str, buffer, memoryview, etc.

A string or other memory buffer containing an entire FITS file. It should be noted that if that memory is read-only (such as a Python string) the returned HDUList's data portions will also be read-only.

kwargs : dict

Optional keyword arguments. See `astropy.io.fits.open()` for details.

Returns**hdul** : HDUList

An HDUList object representing the in-memory FITS file.

index_of (*key*)

Get the index of an HDU from the HDUList.

Parameters**key** : int, str or tuple of (string, int)The key identifying the HDU. If *key* is a tuple, it is of the form (*key*, *ver*) where *ver* is an EXTVER value that must match the HDU being searched for.**Returns****index** : int

The index of the HDU in the HDUList.

info (*output=None*)

Summarize the info of the HDUs in this HDUList.

Note that this function prints its results to the console—it does not return a value.

Parameters**output** : file, bool, optionalA file-like object to write the output to. If *False*, does not output to a file and instead returns a list of tuples representing the HDU info. Writes to `sys.stdout` by default.**insert** (*index, hdu*)Insert an HDU into the HDUList at the given *index*.**Parameters****index** : int

Index before which to insert the new HDU.

hdu : HDU object

The HDU object to insert

readall ()

Read data of all HDUs into memory.

update_extend ()

Make sure that if the primary header needs the keyword EXTEND that it has it and it is correct.

writeto (*fileobj, output_verify='exception', clobber=False, checksum=False*)

Write the HDUList to a new file.

Parameters**fileobj** : file path, file object or file-like object

File to write to. If a file object, must be opened in a writeable mode.

output_verify : strOutput verification option. Must be one of "fix", "silentfix", "ignore", "warn", or "exception". May also be any combination of "fix" or "silentfix" with "+ignore", "+warn", or "+exception" (e.g. ``fix+warn``). See *Verification options* for more info.**clobber** : boolWhen *True*, overwrite the output file if exists.

checksum : bool

When `True` adds both DATASUM and CHECKSUM cards to the headers of all HDU's written to the file.

14.5.3 Header Data Units

The `ImageHDU` and `CompImageHDU` classes are discussed in the section on *Images*.

The `TableHDU` and `BinTableHDU` classes are discussed in the section on *Tables*.

PrimaryHDU

```
class astropy.io.fits.PrimaryHDU (data=None, header=None, do_not_scale_image_data=False, ignore_blank=False, uint=False, scale_back=None)
    Bases: astropy.io.fits.hdu.image._ImageBaseHDU
```

FITS primary HDU class.

Construct a primary HDU.

Parameters

data : array or DELAYED, optional

The data in the HDU.

header : Header instance, optional

The header to be used (as a template). If `header` is `None`, a minimal header will be provided.

do_not_scale_image_data : bool, optional

If `True`, image data is not scaled using BSCALE/BZERO values when read.

ignore_blank : bool, optional

If `True`, the BLANK header keyword will be ignored if present. Otherwise, pixels equal to this value will be replaced with NaNs.

uint : bool, optional

Interpret signed integer data where BZERO is the central value and BSCALE == 1 as unsigned integer data. For example, `int16` data with `BZERO = 32768` and `BSCALE = 1` would be treated as `uint16` data.

scale_back : bool, optional

If `True`, when saving changes to a file that contained scaled image data, restore the data to the original type and reapply the original BSCALE/BZERO values. This could lead to loss of accuracy if scaling back to integer values after performing floating point operations on the data.

```
add_checksum (when=None, override_datasum=False, blocking='standard', checksum_keyword='CHECKSUM', datasum_keyword='DATASUM')
```

Add the CHECKSUM and DATASUM cards to this HDU with the values set to the checksum calculated for the HDU and the data respectively. The addition of the DATASUM card may be overridden.

Parameters

when : str, optional

comment string for the cards; by default the comments will represent the time when the checksum was calculated

override_datasum : bool, optional

add the CHECKSUM card only

blocking : str, optional

“standard” or “nonstandard”, compute sum 2880 bytes at a time, or not

checksum_keyword : str, optional

The name of the header keyword to store the checksum value in; this is typically ‘CHECKSUM’ per convention, but there exist use cases in which a different keyword should be used

datasum_keyword : str, optional

See `checksum_keyword`

Notes

For testing purposes, first call `add_datasum` with a `when` argument, then call `add_checksum` with a `when` argument and `override_datasum` set to `True`. This will provide consistent comments for both cards and enable the generation of a CHECKSUM card with a consistent value.

add_datasum (*when=None, blocking='standard', datasum_keyword='DATASUM'*)

Add the DATASUM card to this HDU with the value set to the checksum calculated for the data.

Parameters

when : str, optional

Comment string for the card that by default represents the time when the checksum was calculated

blocking : str, optional

“standard” or “nonstandard”, compute sum 2880 bytes at a time, or not

datasum_keyword : str, optional

The name of the header keyword to store the datasum value in; this is typically ‘DATASUM’ per convention, but there exist use cases in which a different keyword should be used

Returns

checksum : int

The calculated datasum

Notes

For testing purposes, provide a `when` argument to enable the comment value in the card to remain consistent. This will enable the generation of a CHECKSUM card with a consistent value.

copy ()

Make a copy of the HDU, both header and data are copied.

data

Image/array data as a `ndarray`.

Please remember that the order of axes on an Numpy array are opposite of the order specified in the FITS file. For example for a 2D image the “rows” or y-axis are the first dimension, and the “columns” or x-axis are the second dimension.

If the data is scaled using the BZERO and BSCALE parameters, this attribute returns the data scaled to its physical values unless the file was opened with `do_not_scale_image_data=True`.

filebytes ()

Calculates and returns the number of bytes that this HDU will write to a file.

fileinfo ()

Returns a dictionary detailing information about the locations of this HDU within any associated file. The values are only valid after a read or write of the associated file with no intervening changes to the `HDUList`.

Returns

dict or None

The dictionary details information about the locations of this HDU within an associated file. Returns `None` when the HDU is not associated with a file.

Dictionary contents:

Key	Value
file	File object associated with the HDU
file-mode	Mode in which the file was opened (readonly, copyonwrite, update, append, ostream)
hdrLoc	Starting byte location of header in file
datLoc	Starting byte location of data block in file
datSpan	Data size including padding

classmethod fromstring (data, checksum=False, ignore_missing_end=False, **kwargs)

Creates a new HDU object of the appropriate type from a string containing the HDU’s entire header and, optionally, its data.

Note: When creating a new HDU from a string without a backing file object, the data of that HDU may be read-only. It depends on whether the underlying string was an immutable Python `str/bytes` object, or some kind of read-write memory buffer such as a `memoryview`.

Parameters

data : str, bytearray, memoryview, ndarray

A byte string containing the HDU’s header and data.

checksum : bool, optional

Check the HDU’s checksum and/or datasum.

ignore_missing_end : bool, optional

Ignore a missing end card in the header data. Note that without the end card the end of the header may be ambiguous and resulted in a corrupt HDU. In this case the assumption is that the first 2880 block that does not begin with valid FITS header data is the beginning of the data.

kwargs : optional

May consist of additional keyword arguments specific to an HDU type—these correspond to keywords recognized by the constructors of different HDU classes such as

`PrimaryHDU`, `ImageHDU`, or `BinTableHDU`. Any unrecognized keyword arguments are simply ignored.

classmethod `readfrom` (*fileobj*, *checksum=False*, *ignore_missing_end=False*, ***kwargs*)

Read the HDU from a file. Normally an HDU should be opened with `open()` which reads the entire HDU list in a FITS file. But this method is still provided for symmetry with `writeto()`.

Parameters

fileobj : file object or file-like object

Input FITS file. The file's seek pointer is assumed to be at the beginning of the HDU.

checksum : bool

If `True`, verifies that both `DATASUM` and `CHECKSUM` card values (when present in the HDU header) match the header and data of all HDU's in the file.

ignore_missing_end : bool

Do not issue an exception when opening a file that is missing an `END` card in the last header.

req_cards (*keyword*, *pos*, *test*, *fix_value*, *option*, *errlist*)

Check the existence, location, and value of a required `Card`.

Parameters

keyword : str

The keyword to validate

pos : int, callable

If an `int`, this specifies the exact location this card should have in the header. Remember that Python is zero-indexed, so this means `pos=0` requires the card to be the first card in the header. If given a callable, it should take one argument—the actual position of the keyword—and return `True` or `False`. This can be used for custom evaluation. For example if `pos=lambda idx: idx > 10` this will check that the keyword's index is greater than 10.

test : callable

This should be a callable (generally a function) that is passed the value of the given keyword and returns `True` or `False`. This can be used to validate the value associated with the given keyword.

fix_value : str, int, float, complex, bool, None

A valid value for a FITS keyword to use if the given `test` fails to replace an invalid value. In other words, this provides a default value to use as a replacement if the keyword's current value is invalid. If `None`, there is no replacement value and the keyword is unfixable.

option : str

Output verification option. Must be one of `"fix"`, `"silentfix"`, `"ignore"`, `"warn"`, or `"exception"`. May also be any combination of `"fix"` or `"silentfix"` with `"+ignore"`, `+warn`, or `+exception`" (e.g. ```fix+warn```). See [Verification options](#) for more info.

errlist : list

A list of validation errors already found in the FITS file; this is used primarily for the validation system to collect errors across multiple HDUs and multiple calls to `req_cards`.

Notes

If `pos=None`, the card can be anywhere in the header. If the card does not exist, the new card will have the `fix_value` as its value when created. Also check the card's value by using the `test` argument.

run_option (*option=u'warn', err_text=u'', fix_text=u'Fixed.', fix=None, fixable=True*)

Execute the verification with selected option.

scale (*type=None, option='old', bscale=1, bzero=0*)

Scale image data by using `BSCALE/BZERO`.

Call to this method will scale `data` and update the keywords of `BSCALE` and `BZERO` in the HDU's header. This method should only be used right before writing to the output file, as the data will be scaled and is therefore not very usable after the call.

Parameters

type : str, optional

destination data type, use a string representing a numpy dtype name, (e.g. `'uint8'`, `'int16'`, `'float32'` etc.). If is `None`, use the current data type.

option : str

How to scale the data: if `"old"`, use the original `BSCALE` and `BZERO` values when the data was read/created. If `"minmax"`, use the minimum and maximum of the data to scale. The option will be overwritten by any user specified `bscale/bzero` values.

bscale, bzero : int, optional

User-specified `BSCALE` and `BZERO` values

section

Access a section of the image array without loading the entire array into memory. The `Section` object returned by this attribute is not meant to be used directly by itself. Rather, slices of the section return the appropriate slice of the data, and loads *only* that section into memory.

Sections are mostly obsoleted by memmap support, but should still be used to deal with very large scaled images. See the *Data Sections* section of the PyFITS documentation for more details.

shape

Shape of the image array—should be equivalent to `self.data.shape`.

size

Size (in bytes) of the data portion of the HDU.

update_ext_name (**args, **kwargs*)

Deprecated since version 0.3: The `update_ext_name` function will be deprecated in a future version. Use the `.name` attribute or `Header.set` instead.

Update the extension name associated with the HDU.

If the keyword already exists in the Header, it's value and/or comment will be updated. If it does not exist, a new card will be created and it will be placed before or after the specified location. If no `before` or `after` is specified, it will be appended at the end.

Parameters

value : str

Value to be used for the new extension name

comment : str, optional

To be used for updating, default=`None`.

before : str or int, optional

Name of the keyword, or index of the `Card` before which the new card will be placed in the Header. The argument `before` takes precedence over `after` if both are specified.

after : str or int, optional

Name of the keyword, or index of the `Card` after which the new card will be placed in the Header

savecomment : bool, optional

When `True`, preserve the current comment for an existing keyword. The argument `savecomment` takes precedence over `comment` if both specified. If `comment` is not specified then the current comment will automatically be preserved.

update_ext_version (**args, **kwargs*)

Deprecated since version 0.3: The `update_ext_version` function will be deprecated in a future version. Use the `.ver` attribute or `Header.set` instead.

Update the extension version associated with the HDU.

If the keyword already exists in the Header, it's value and/or comment will be updated. If it does not exist, a new card will be created and it will be placed before or after the specified location. If no `before` or `after` is specified, it will be appended at the end.

Parameters

value : str

Value to be used for the new extension version

comment : str, optional

To be used for updating; default=None.

before : str or int, optional

Name of the keyword, or index of the `Card` before which the new card will be placed in the Header. The argument `before` takes precedence over `after` if both are specified.

after : str or int, optional

Name of the keyword, or index of the `Card` after which the new card will be placed in the Header.

savecomment : bool, optional

When `True`, preserve the current comment for an existing keyword. The argument `savecomment` takes precedence over `comment` if both specified. If `comment` is not specified then the current comment will automatically be preserved.

verify (*option=u'warn'*)

Verify all values in the instance.

Parameters

option : str

Output verification option. Must be one of "fix", "silentfix", "ignore", "warn", or "exception". May also be any combination of "fix" or "silentfix" with "+ignore", "+warn, or "+exception" (e.g. ``"fix+warn"). See [Verification options](#) for more info.

verify_checksum (*blocking='standard'*)

Verify that the value in the CHECKSUM keyword matches the value calculated for the current HDU CHECKSUM.

blocking: str, optional

“standard” or “nonstandard”, compute sum 2880 bytes at a time, or not

Returns

valid : int

- 0 - failure
- 1 - success
- 2 - no CHECKSUM keyword present

verify_datasum (*blocking='standard'*)

Verify that the value in the DATASUM keyword matches the value calculated for the DATASUM of the current HDU data.

blocking: str, optional

“standard” or “nonstandard”, compute sum 2880 bytes at a time, or not

Returns

valid : int

- 0 - failure
- 1 - success
- 2 - no DATASUM keyword present

writeto (*name, output_verify='exception', clobber=False, checksum=False*)

Write the HDU to a new file. This is a convenience method to provide a user easier output interface if only one HDU needs to be written to a file.

Parameters

name : file path, file object or file-like object

Output FITS file. If the file object is already opened, it must be opened in a writeable mode.

output_verify : str

Output verification option. Must be one of "fix", "silentfix", "ignore", "warn", or "exception". May also be any combination of "fix" or "silentfix" with "+ignore", +warn, or +exception" (e.g. ``"fix+warn"). See [Verification options](#) for more info.

clobber : bool

Overwrite the output file if exists.

checksum : bool

When `True` adds both DATASUM and CHECKSUM cards to the header of the HDU when written to the file.

GroupsHDU

class `astropy.io.fits.GroupsHDU` (*data=None, header=None*)

Bases: `astropy.io.fits.PrimaryHDU`, `astropy.io.fits.hdu.table._TableLikeHDU`

FITS Random Groups HDU class.

See the *Random Access Groups* section in the PyFITS documentation for more details on working with this type of HDU.

add_checksum (*when=None*, *override_datasum=False*, *blocking='standard'*, *checksum_keyword='CHECKSUM'*, *datasum_keyword='DATASUM'*)

Add the CHECKSUM and DATASUM cards to this HDU with the values set to the checksum calculated for the HDU and the data respectively. The addition of the DATASUM card may be overridden.

Parameters

when : str, optional

comment string for the cards; by default the comments will represent the time when the checksum was calculated

override_datasum : bool, optional

add the CHECKSUM card only

blocking : str, optional

“standard” or “nonstandard”, compute sum 2880 bytes at a time, or not

checksum_keyword : str, optional

The name of the header keyword to store the checksum value in; this is typically ‘CHECKSUM’ per convention, but there exist use cases in which a different keyword should be used

datasum_keyword : str, optional

See `checksum_keyword`

Notes

For testing purposes, first call `add_datasum` with a `when` argument, then call `add_checksum` with a `when` argument and `override_datasum` set to `True`. This will provide consistent comments for both cards and enable the generation of a CHECKSUM card with a consistent value.

add_datasum (*when=None*, *blocking='standard'*, *datasum_keyword='DATASUM'*)

Add the DATASUM card to this HDU with the value set to the checksum calculated for the data.

Parameters

when : str, optional

Comment string for the card that by default represents the time when the checksum was calculated

blocking : str, optional

“standard” or “nonstandard”, compute sum 2880 bytes at a time, or not

datasum_keyword : str, optional

The name of the header keyword to store the datasum value in; this is typically ‘DATASUM’ per convention, but there exist use cases in which a different keyword should be used

Returns

checksum : int

The calculated datasum

Notes

For testing purposes, provide a `when` argument to enable the comment value in the card to remain consistent. This will enable the generation of a `CHECKSUM` card with a consistent value.

`copy()`

Make a copy of the HDU, both header and data are copied.

`data`

The data of a random group FITS file will be like a binary table's data.

`filebytes()`

Calculates and returns the number of bytes that this HDU will write to a file.

`fileinfo()`

Returns a dictionary detailing information about the locations of this HDU within any associated file. The values are only valid after a read or write of the associated file with no intervening changes to the `HDUList`.

Returns

dict or None

The dictionary details information about the locations of this HDU within an associated file. Returns `None` when the HDU is not associated with a file.

Dictionary contents:

Key	Value
<code>file</code>	File object associated with the HDU
<code>file-mode</code>	Mode in which the file was opened (readonly, copyonwrite, update, append, ostream)
<code>hdrLoc</code>	Starting byte location of header in file
<code>datLoc</code>	Starting byte location of data block in file
<code>datSpan</code>	Data size including padding

`classmethod from_columns` (*columns, header=None, nrows=0, fill=False, **kwargs*)

Given either a `ColDefs` object, a sequence of `Column` objects, or another table HDU or table data (a `FITS_rec` or multi-field `numpy.ndarray` or `numpy.recarray` object, return a new table HDU of the class this method was called on using the column definition from the input.

This is an alternative to the now deprecated `new_table` function, and otherwise accepts the same arguments. See also `FITS_rec.from_columns`.

Parameters

columns : sequence of `Column`, `ColDefs`, or other

The columns from which to create the table data, or an object with a column-like structure from which a `ColDefs` can be instantiated. This includes an existing `BinTableHDU` or `TableHDU`, or a `numpy.recarray` to give some examples.

If these columns have data arrays attached that data may be used in initializing the new table. Otherwise the input columns will be used as a template for a new table with the requested number of rows.

header : `Header`

An optional `Header` object to instantiate the new HDU yet. Header keywords specifically related to defining the table structure (such as the “`TXXXn`” keywords like `TTYPEn`) will be overridden by the supplied column definitions, but all other informational and data model-specific keywords are kept.

nrows : int

Number of rows in the new table. If the input columns have data associated with them, the size of the largest input column is used. Otherwise the default is 0.

fill : bool

If `True`, will fill all cells with zeros or blanks. If `False`, copy the data from input, undefined cells will still be filled with zeros/blanks.

Notes

Any additional keyword arguments accepted by the HDU class's `__init__` may also be passed in as keyword arguments.

classmethod `fromstring` (*data*, *checksum=False*, *ignore_missing_end=False*, ***kwargs*)

Creates a new HDU object of the appropriate type from a string containing the HDU's entire header and, optionally, its data.

Note: When creating a new HDU from a string without a backing file object, the data of that HDU may be read-only. It depends on whether the underlying string was an immutable Python `str/bytes` object, or some kind of read-write memory buffer such as a `memoryview`.

Parameters

data : str, bytearray, memoryview, ndarray

A byte string containing the HDU's header and data.

checksum : bool, optional

Check the HDU's checksum and/or datasum.

ignore_missing_end : bool, optional

Ignore a missing end card in the header data. Note that without the end card the end of the header may be ambiguous and result in a corrupt HDU. In this case the assumption is that the first 2880 block that does not begin with valid FITS header data is the beginning of the data.

kwargs : optional

May consist of additional keyword arguments specific to an HDU type—these correspond to keywords recognized by the constructors of different HDU classes such as `PrimaryHDU`, `ImageHDU`, or `BinTableHDU`. Any unrecognized keyword arguments are simply ignored.

parnames

The names of the group parameters as described by the header.

classmethod `readfrom` (*fileobj*, *checksum=False*, *ignore_missing_end=False*, ***kwargs*)

Read the HDU from a file. Normally an HDU should be opened with `open()` which reads the entire HDU list in a FITS file. But this method is still provided for symmetry with `writeto()`.

Parameters

fileobj : file object or file-like object

Input FITS file. The file's seek pointer is assumed to be at the beginning of the HDU.

checksum : bool

If `True`, verifies that both `DATASUM` and `CHECKSUM` card values (when present in the HDU header) match the header and data of all HDU's in the file.

ignore_missing_end : bool

Do not issue an exception when opening a file that is missing an END card in the last header.

req_cards (*keyword, pos, test, fix_value, option, errlist*)

Check the existence, location, and value of a required [Card](#).

Parameters

keyword : str

The keyword to validate

pos : int, callable

If an `int`, this specifies the exact location this card should have in the header. Remember that Python is zero-indexed, so this means `pos=0` requires the card to be the first card in the header. If given a callable, it should take one argument—the actual position of the keyword—and return `True` or `False`. This can be used for custom evaluation. For example if `pos=lambda idx: idx > 10` this will check that the keyword's index is greater than 10.

test : callable

This should be a callable (generally a function) that is passed the value of the given keyword and returns `True` or `False`. This can be used to validate the value associated with the given keyword.

fix_value : str, int, float, complex, bool, None

A valid value for a FITS keyword to use if the given `test` fails to replace an invalid value. In other words, this provides a default value to use as a replacement if the keyword's current value is invalid. If `None`, there is no replacement value and the keyword is unfixable.

option : str

Output verification option. Must be one of "fix", "silentfix", "ignore", "warn", or "exception". May also be any combination of "fix" or "silentfix" with "+ignore", "+warn, or "+exception" (e.g. `fix+warn`). See [Verification options](#) for more info.

errlist : list

A list of validation errors already found in the FITS file; this is used primarily for the validation system to collect errors across multiple HDUs and multiple calls to `req_cards`.

Notes

If `pos=None`, the card can be anywhere in the header. If the card does not exist, the new card will have the `fix_value` as its value when created. Also check the card's value by using the `test` argument.

run_option (*option=u'warn', err_text=u'', fix_text=u'Fixed.', fix=None, fixable=True*)

Execute the verification with selected option.

scale (*type=None, option='old', bscale=1, bzero=0*)

Scale image data by using BSCALE/BZERO.

Call to this method will scale `data` and update the keywords of BSCALE and BZERO in the HDU's header. This method should only be used right before writing to the output file, as the data will be scaled and is therefore not very usable after the call.

Parameters**type** : str, optional

destination data type, use a string representing a numpy dtype name, (e.g. 'uint8', 'int16', 'float32' etc.). If is `None`, use the current data type.

option : str

How to scale the data: if "old", use the original BSCALE and BZERO values when the data was read/created. If "minmax", use the minimum and maximum of the data to scale. The option will be overwritten by any user specified `b scale/bzero` values.

b scale, bzero : int, optional

User-specified BSCALE and BZERO values

section

Access a section of the image array without loading the entire array into memory. The `Section` object returned by this attribute is not meant to be used directly by itself. Rather, slices of the section return the appropriate slice of the data, and loads *only* that section into memory.

Sections are mostly obsoleted by memmap support, but should still be used to deal with very large scaled images. See the *Data Sections* section of the PyFITS documentation for more details.

shape

Shape of the image array—should be equivalent to `self.data.shape`.

size

Returns the size (in bytes) of the HDU's data part.

update_ext_name (*args, **kwargs)

Deprecated since version 0.3: The `update_ext_name` function will be deprecated in a future version. Use the `.name` attribute or `Header.set` instead.

Update the extension name associated with the HDU.

If the keyword already exists in the Header, it's value and/or comment will be updated. If it does not exist, a new card will be created and it will be placed before or after the specified location. If no `before` or `after` is specified, it will be appended at the end.

Parameters**value** : str

Value to be used for the new extension name

comment : str, optional

To be used for updating, default=None.

before : str or int, optional

Name of the keyword, or index of the `Card` before which the new card will be placed in the Header. The argument `before` takes precedence over `after` if both are specified.

after : str or int, optional

Name of the keyword, or index of the `Card` after which the new card will be placed in the Header

savecomment : bool, optional

When `True`, preserve the current comment for an existing keyword. The argument `savecomment` takes precedence over `comment` if both specified. If `comment` is not specified then the current comment will automatically be preserved.

update_ext_version (*args, **kwargs)

Deprecated since version 0.3: The `update_ext_version` function will be deprecated in a future version. Use the `.ver` attribute or `Header.set` instead.

Update the extension version associated with the HDU.

If the keyword already exists in the Header, its value and/or comment will be updated. If it does not exist, a new card will be created and it will be placed before or after the specified location. If no `before` or `after` is specified, it will be appended at the end.

Parameters

value : str

Value to be used for the new extension version

comment : str, optional

To be used for updating; default=None.

before : str or int, optional

Name of the keyword, or index of the `Card` before which the new card will be placed in the Header. The argument `before` takes precedence over `after` if both are specified.

after : str or int, optional

Name of the keyword, or index of the `Card` after which the new card will be placed in the Header.

savecomment : bool, optional

When `True`, preserve the current comment for an existing keyword. The argument `savecomment` takes precedence over `comment` if both specified. If `comment` is not specified then the current comment will automatically be preserved.

verify (option=*u'warn'*)

Verify all values in the instance.

Parameters

option : str

Output verification option. Must be one of "fix", "silentfix", "ignore", "warn", or "exception". May also be any combination of "fix" or "silentfix" with "+ignore", "+warn, or +exception" (e.g. `fix+warn`). See [Verification options](#) for more info.

verify_checksum (blocking=*'standard'*)

Verify that the value in the CHECKSUM keyword matches the value calculated for the current HDU CHECKSUM.

blocking: str, optional

“standard” or “nonstandard”, compute sum 2880 bytes at a time, or not

Returns

valid : int

- 0 - failure
- 1 - success
- 2 - no CHECKSUM keyword present

verify_datasum (*blocking='standard'*)

Verify that the value in the DATASUM keyword matches the value calculated for the DATASUM of the current HDU data.

blocking: str, optional

“standard” or “nonstandard”, compute sum 2880 bytes at a time, or not

Returns

valid : int

- 0 - failure
- 1 - success
- 2 - no DATASUM keyword present

writeto (*name, output_verify='exception', clobber=False, checksum=False*)

Write the HDU to a new file. This is a convenience method to provide a user easier output interface if only one HDU needs to be written to a file.

Parameters

name : file path, file object or file-like object

Output FITS file. If the file object is already opened, it must be opened in a writeable mode.

output_verify : str

Output verification option. Must be one of "fix", "silentfix", "ignore", "warn", or "exception". May also be any combination of "fix" or "silentfix" with "+ignore", "+warn", or "+exception" (e.g. ``"fix+warn"). See [Verification options](#) for more info.

clobber : bool

Overwrite the output file if exists.

checksum : bool

When `True` adds both DATASUM and CHECKSUM cards to the header of the HDU when written to the file.

GroupData

class `astropy.io.fits.GroupData`

Bases: `astropy.io.fits.FITS_rec`

Random groups data object.

Allows structured access to FITS Group data in a manner analogous to tables.

data

The raw group data represented as a multi-dimensional `numpy.ndarray` array.

par (*parname*)

Get the group parameter values.

Group

class `astropy.io.fits.Group` (*input*, *row=0*, *start=None*, *end=None*, *step=None*, *base=None*)

Bases: `astropy.io.fits.FITS_record`

One group of the random group data.

par (*parname*)

Get the group parameter value.

setpar (*parname*, *value*)

Set the group parameter value.

StreamingHDU

class `astropy.io.fits.StreamingHDU` (*name*, *header*)

Bases: `object`

A class that provides the capability to stream data to a FITS file instead of requiring data to all be written at once.

The following pseudocode illustrates its use:

```
header = astropy.io.fits.Header()

for all the cards you need in the header:
    header[key] = (value, comment)

shdu = astropy.io.fits.StreamingHDU('filename.fits', header)

for each piece of data:
    shdu.write(data)

shdu.close()
```

Construct a `StreamingHDU` object given a file name and a header.

Parameters

name : file path, file object, or file like object

The file to which the header and data will be streamed. If opened, the file object must be opened in a writeable binary mode such as 'wb' or 'ab+'.

header : `Header` instance

The header object associated with the data to be written to the file.

Notes

The file will be opened and the header appended to the end of the file. If the file does not already exist, it will be created, and if the header represents a Primary header, it will be written to the beginning of the file. If the file does not exist and the provided header is not a Primary header, a default Primary HDU will be inserted at the beginning of the file and the provided header will be added as the first extension. If the file does already exist, but the provided header represents a Primary header, the header will be modified to an image extension header and appended to the end of the file.

close ()

Close the physical FITS file.

size

Return the size (in bytes) of the data portion of the HDU.

write (*data*)

Write the given data to the stream.

Parameters

data : ndarray

Data to stream to the file.

Returns

writecomplete : int

Flag that when `True` indicates that all of the required data has been written to the stream.

Notes

Only the amount of data specified in the header provided to the class constructor may be written to the stream. If the provided data would cause the stream to overflow, an `IOError` exception is raised and the data is not written. Once sufficient data has been written to the stream to satisfy the amount specified in the header, the stream is padded to fill a complete FITS block and no more data will be accepted. An attempt to write more data after the stream has been filled will raise an `IOError` exception. If the dtype of the input data does not match what is expected by the header, a `exceptions.TypeError` exception is raised.

14.5.4 Headers

Header

class `astropy.io.fits.Header` (*cards=[]*, *txtfile=None*)

Bases: `object`

FITS header class. This class exposes both a dict-like interface and a list-like interface to FITS headers.

The header may be indexed by keyword and, like a dict, the associated value will be returned. When the header contains cards with duplicate keywords, only the value of the first card with the given keyword will be returned. It is also possible to use a 2-tuple as the index in the form (keyword, n)—this returns the n-th value with that keyword, in the case where there are duplicate keywords.

For example:

```
>>> header['NAXIS']
0
>>> header[('FOO', 1)] # Return the value of the second FOO keyword
'foo'
```

The header may also be indexed by card number:

```
>>> header[0] # Return the value of the first card in the header
'T'
```

Commentary keywords such as `HISTORY` and `COMMENT` are special cases: When indexing the `Header` object with either `'HISTORY'` or `'COMMENT'` a list of all the `HISTORY/COMMENT` values is returned:

```
>>> header['HISTORY']
This is the first history entry in this header.
This is the second history entry in this header.
...

```

See the Astropy documentation for more details on working with headers.

Construct a `Header` from an iterable and/or text file.

Parameters

cards : A list of `Card` objects, optional

The cards to initialize the header with.

txtfile : file path, file object or file-like object, optional

Input ASCII header parameters file (**Deprecated**) Use the `Header.fromfile` classmethod instead.

add_blank (*value='', before=None, after=None*)

Add a blank card.

Parameters

value : str, optional

Text to be added.

before : str or int, optional

Same as in `Header.update`

after : str or int, optional

Same as in `Header.update`

add_comment (*value, before=None, after=None*)

Add a COMMENT card.

Parameters

value : str

Text to be added.

before : str or int, optional

Same as in `Header.update`

after : str or int, optional

Same as in `Header.update`

add_history (*value, before=None, after=None*)

Add a HISTORY card.

Parameters

value : str

History text to be added.

before : str or int, optional

Same as in `Header.update`

after : str or int, optional

Same as in `Header.update`

append (*card=None, useblanks=True, bottom=False, end=False*)

Appends a new keyword+value card to the end of the Header, similar to `list.append`.

By default if the last cards in the Header have commentary keywords, this will append the new keyword before the commentary (unless the new keyword is also commentary).

Also differs from `list.append` in that it can be called with no arguments: In this case a blank card is appended to the end of the Header. In the case all the keyword arguments are ignored.

Parameters

card : str, tuple

A keyword or a (keyword, value, [comment]) tuple representing a single header card; the comment is optional in which case a 2-tuple may be used

useblanks : bool, optional

If there are blank cards at the end of the Header, replace the first blank card so that the total number of cards in the Header does not increase. Otherwise preserve the number of blank cards.

bottom : bool, optional

If True, instead of appending after the last non-commentary card, append after the last non-blank card.

end : bool, optional

If True, ignore the useblanks and bottom options, and append at the very end of the Header.

ascard

Deprecated since version 0.1: The `ascard` function is deprecated and may be removed in a future version. Use the `cards` attribute instead.

Returns a `CardList` object wrapping this Header; provided for backwards compatibility for the old API (where Headers had an underlying `CardList`).

cards

The underlying physical cards that make up this Header; it can be looked at, but it should not be modified directly.

clear()

Remove all cards from the header.

comments

View the comments associated with each keyword, if any.

For example, to see the comment on the NAXIS keyword:

```
>>> header.comments['NAXIS']
number of data axes
```

Comments can also be updated through this interface:

```
>>> header.comments['NAXIS'] = 'Number of data axes'
```

copy (*strip=False*)

Make a copy of the `Header`.

Parameters

strip : bool, optional

If `True`, strip any headers that are specific to one of the standard HDU types, so that this header can be used in a different HDU.

Returns

header

A new `Header` instance.

count (*keyword*)

Returns the count of the given keyword in the header, similar to `list.count` if the `Header` object is treated as a list of keywords.

Parameters

keyword : str

The keyword to count instances of in the header

extend (*cards*, *strip=True*, *unique=False*, *update=False*, *update_first=False*, *useblanks=True*, *bottom=False*, *end=False*)

Appends multiple keyword+value cards to the end of the header, similar to `list.extend`.

Parameters

cards : iterable

An iterable of (keyword, value, [comment]) tuples; see `Header.append`.

strip : bool, optional

Remove any keywords that have meaning only to specific types of HDUs, so that only more general keywords are added from extension `Header` or `Card` list (default: `True`).

unique : bool, optional

If `True`, ensures that no duplicate keywords are appended; keywords already in this header are simply discarded. The exception is commentary keywords (`COMMENT`, `HISTORY`, etc.): they are only treated as duplicates if their values match.

update : bool, optional

If `True`, update the current header with the values and comments from duplicate keywords in the input header. This supercedes the `unique` argument. Commentary keywords are treated the same as if `unique=True`.

update_first : bool, optional

If the first keyword in the header is 'SIMPLE', and the first keyword in the input header is 'XTENSION', the 'SIMPLE' keyword is replaced by the 'XTENSION' keyword. Likewise if the first keyword in the header is 'XTENSION' and the first keyword in the input header is 'SIMPLE', the 'XTENSION' keyword is replaced by the 'SIMPLE' keyword. This behavior is otherwise dumb as to whether or not the resulting header is a valid primary or extension header. This is mostly provided to support backwards compatibility with the old `Header.fromTxtFile()` method, and only applies if `update=True`.

useblanks, bottom, end : bool, optional

These arguments are passed to `Header.append()` while appending new cards to the header.

fromTxtFile (**args*, ***kwargs*)

Deprecated since version 0.1: This is equivalent to `self.extend(Header.fromtextfile(fileobj), update=True, update_first=True)`. Note that there there is no direct equivalent to the `replace=True` option since `Header.fromtextfile()` returns a new `Header` instance.

Input the header parameters from an ASCII file.

The input header cards will be used to update the current header. Therefore, when an input card key matches a card key that already exists in the header, that card will be updated in place. Any input cards that do not already exist in the header will be added. Cards will not be deleted from the header.

Parameters

fileobj : file path, file object or file-like object

Input header parameters file.

replace : bool, optional

When `True`, indicates that the entire header should be replaced with the contents of the ASCII file instead of just updating the current header.

classmethod fromfile (*fileobj*, *sep=''*, *endcard=True*, *padding=True*)

Similar to `Header.fromstring()`, but reads the header string from a given file-like object or file-name.

Parameters

fileobj : str, file-like

A filename or an open file-like object from which a FITS header is to be read. For open file handles the file pointer must be at the beginning of the header.

sep : str, optional

The string separating cards from each other, such as a newline. By default there is no card separator (as is the case in a raw FITS file).

endcard : bool, optional

If `True` (the default) the header must end with an END card in order to be considered valid. If an END card is not found an `IOError` is raised.

padding : bool, optional

If `True` (the default) the header will be required to be padded out to a multiple of 2880, the FITS header block size. Otherwise any padding, or lack thereof, is ignored.

Returns

header

A new `Header` instance.

classmethod fromkeys (*iterable*, *value=None*)

Similar to `dict.fromkeys()` –creates a new `Header` from an iterable of keywords and an optional default value.

This method is not likely to be particularly useful for creating real world FITS headers, but it is useful for testing.

Parameters

iterable

Any iterable that returns strings representing FITS keywords.

value : optional

A default value to assign to each keyword; must be a valid type for FITS keywords.

Returns

header

A new `Header` instance.

classmethod fromstring (*data*, *sep*='')

Creates an HDU header from a byte string containing the entire header data.

Parameters

data : str

String containing the entire header.

sep : str, optional

The string separating cards from each other, such as a newline. By default there is no card separator (as is the case in a raw FITS file).

Returns

header

A new `Header` instance.

classmethod fromtextfile (*fileobj*, *endcard*=False)

Equivalent to:

```
>>> Header.fromfile(fileobj, sep='\n', endcard=False,
...                 padding=False)
```

get (*key*, *default*=None)

Similar to `dict.get()` –returns the value associated with keyword in the header, or a default value if the keyword is not found.

Parameters

key : str

A keyword that may or may not be in the header.

default : optional

A default value to return if the keyword is not found in the header.

Returns

value

The value associated with the given keyword, or the default value if the keyword is not in the header.

get_comment (**args*, ***kwargs*)

Deprecated since version 0.1: The `get_comment` function is deprecated and may be removed in a future version. Use `header['COMMENT']` instead.

Get all comment cards as a list of string texts.

get_history (**args*, ***kwargs*)

Deprecated since version 0.1: The `get_history` function is deprecated and may be removed in a future version. Use `header['HISTORY']` instead.

Get all history cards as a list of string texts.

index (*keyword*, *start*=None, *stop*=None)

Returns the index of the first instance of the given keyword in the header, similar to `list.index` if the Header object is treated as a list of keywords.

Parameters

keyword : str

The keyword to look up in the list of all keywords in the header

start : int, optional

The lower bound for the index

stop : int, optional

The upper bound for the index

insert (*key, card, useblanks=True, after=False*)

Inserts a new keyword+value card into the Header at a given location, similar to `list.insert`.

Parameters

key : int, str, or tuple

The index into the the list of header keywords before which the new keyword should be inserted, or the name of a keyword before which the new keyword should be inserted. Can also accept a (keyword, index) tuple for inserting around duplicate keywords.

card : str, tuple

A keyword or a (keyword, value, [comment]) tuple; see `Header.append`

useblanks : bool, optional

If there are blank cards at the end of the Header, replace the first blank card so that the total number of cards in the Header does not increase. Otherwise preserve the number of blank cards.

after : bool, optional

If set to `True`, insert *after* the specified index or keyword, rather than before it. Defaults to `False`.

items ()

Like `dict.items()`.

iteritems ()

Like `dict.iteritems()`.

iterkeys ()

Like `dict.iterkeys()`—iterating directly over the `Header` instance has the same behavior.

intervalues ()

Like `dict.intervalues()`.

keys ()

Return a list of keywords in the header in the order they appear—like `dict.keys()` but ordered.

pop (*args)

Works like `list.pop()` if no arguments or an index argument are supplied; otherwise works like `dict.pop()`.

popitem ()

Similar to `dict.popitem()`.

remove (keyword)

Removes the first instance of the given keyword from the header similar to `list.remove` if the Header object is treated as a list of keywords.

Parameters

keyword : str

The keyword of which to remove the first instance in the header

rename_key (*args, **kwargs)

Deprecated since version 0.1: The `rename_key` function is deprecated and may be removed in a future version. Use `Header.rename_keyword()` instead.

rename_keyword (*oldkeyword, newkeyword, force=False*)

Rename a card's keyword in the header.

Parameters

oldkeyword : str or int

Old keyword or card index

newkeyword : str

New keyword

force : bool, optional

When `True`, if the new keyword already exists in the header, force the creation of a duplicate keyword. Otherwise a `ValueError` is raised.

set (*keyword, value=None, comment=None, before=None, after=None*)

Set the value and/or comment and/or position of a specified keyword.

If the keyword does not already exist in the header, a new keyword is created in the specified position, or appended to the end of the header if no position is specified.

This method is similar to `Header.update()` prior to PyFITS 3.1.

Note: It should be noted that `header.set(keyword, value)` and `header.set(keyword, value, comment)` are equivalent to `header[keyword] = value` and `header[keyword] = (value, comment)` respectfully.

New keywords can also be inserted relative to existing keywords using, for example:

```
>>> header.insert('NAXIS1', ('NAXIS', 2, 'Number of axes'))
```

to insert before an existing keyword, or:

```
>>> header.insert('NAXIS', ('NAXIS1', 4096), after=True)
```

to insert after an existing keyword.

The the only advantage of using `Header.set()` is that it easily replaces the old usage of `Header.update()` both conceptually and in terms of function signature.

Parameters

keyword : str

A header keyword

value : str, optional

The value to set for the given keyword; if `None` the existing value is kept, but `''` may be used to set a blank value

comment : str, optional

The comment to set for the given keyword; if `None` the existing comment is kept, but `''` may be used to set a blank comment

before : str, int, optional

Name of the keyword, or index of the `Card` before which this card should be located in the header. The argument `before` takes precedence over `after` if both specified.

after : str, int, optional

Name of the keyword, or index of the `Card` after which this card should be located in the header.

setdefault (*key*, *default=None*)

Similar to `dict.setdefault()`.

toTxtFile (**args*, ***kwargs*)

Deprecated since version 0.1: The `toTxtFile` function is deprecated and may be removed in a future version. Use `Header.totextfile()` instead.

Output the header parameters to a file in ASCII format.

Parameters

fileobj : file path, file object or file-like object

Output header parameters file.

clobber : bool

When `True`, overwrite the output file if it exists.

tofile (*fileobj*, *sep=''*, *endcard=True*, *padding=True*, *clobber=False*)

Writes the header to file or file-like object.

By default this writes the header exactly as it would be written to a FITS file, with the END card included and padding to the next multiple of 2880 bytes. However, aspects of this may be controlled.

Parameters

fileobj : str, file, optional

Either the pathname of a file, or an open file handle or file-like object

sep : str, optional

The character or string with which to separate cards. By default there is no separator, but one could use `'\\n'`, for example, to separate each card with a new line

endcard : bool, optional

If `True` (default) adds the END card to the end of the header string

padding : bool, optional

If `True` (default) pads the string with spaces out to the next multiple of 2880 characters

clobber : bool, optional

If `True`, overwrites the output file if it already exists

tostring (*sep=''*, *endcard=True*, *padding=True*)

Returns a string representation of the header.

By default this uses no separator between cards, adds the END card, and pads the string with spaces to the next multiple of 2880 bytes. That is, it returns the header exactly as it would appear in a FITS file.

Parameters

sep : str, optional

The character or string with which to separate cards. By default there is no separator, but one could use `'\\n'`, for example, to separate each card with a new line

endcard : bool, optional

If `True` (default) adds the END card to the end of the header string

padding : bool, optional

If True (default) pads the string with spaces out to the next multiple of 2880 characters

Returns

`s` : str

A string representing a FITS header.

totextfile (*fileobj*, *endcard=False*, *clobber=False*)

Equivalent to:

```
>>> Header.tofile(fileobj, sep='\n', endcard=False,
...               padding=False, clobber=clobber)
```

update (**args*, ***kwargs*)

Update the Header with new keyword values, updating the values of existing keywords and appending new keywords otherwise; similar to `dict.update`.

`update` accepts either a dict-like object or an iterable. In the former case the keys must be header keywords and the values may be either scalar values or (value, comment) tuples. In the case of an iterable the items must be (keyword, value) tuples or (keyword, value, comment) tuples.

Arbitrary arguments are also accepted, in which case the `update()` is called again with the `kwargs` dict as its only argument. That is,

```
>>> header.update(NAXIS1=100, NAXIS2=100)
```

is equivalent to:

```
header.update({'NAXIS1': 100, 'NAXIS2': 100})
```

Warning: As this method works similarly to `dict.update` it is very different from the `Header.update()` method in PyFITS versions prior to 3.1.0. However, support for the old API is also maintained for backwards compatibility. If `update()` is called with at least two positional arguments then it can be assumed that the old API is being used. Use of the old API should be considered **deprecated**. Most uses of the old API can be replaced as follows:

- Replace

```
header.update(keyword, value)
```

with

```
header[keyword] = value
```

- Replace

```
header.update(keyword, value, comment=comment)
```

with

```
header[keyword] = (value, comment)
```

- Replace

```
header.update(keyword, value, before=before_keyword)
```

with

```
header.insert(before_keyword, (keyword, value))
```

- Replace

```
header.update(keyword, value, after=after_keyword)
```

with

```
header.insert(after_keyword, (keyword, value),
              after=True)
```

See also `Header.set()` which is a new method that provides an interface similar to the old `Header.update()` and may help make transition a little easier.

For reference, the old documentation for the old `Header.update()` is provided below:

Update one header card.

If the keyword already exists, its value and/or comment will be updated. If it does not exist, a new card will be created and it will be placed before or after the specified location. If no `before` or `after` is specified, it will be appended at the end.

Parameters

key : str

keyword

value : str

value to be used for updating

comment : str, optional

to be used for updating, default=None.

before : str, int, optional

name of the keyword, or index of the `Card` before which the new card will be placed. The argument `before` takes precedence over `after` if both specified.

after : str, int, optional

name of the keyword, or index of the `Card` after which the new card will be placed.

savecomment : bool, optional

When `True`, preserve the current comment for an existing keyword. The argument `savecomment` takes precedence over `comment` if both specified. If `comment` is not specified then the current comment will automatically be preserved.

values ()

Returns a list of the values of all cards in the header.

14.5.5 Cards

Card

class `astropy.io.fits.Card` (*keyword=None, value=None, comment=None, **kwargs*)

Bases: `astropy.io.fits.verify._Verify`

ascardimage (**args, **kwargs*)

Deprecated since version 0.1: The `ascardimage` function is deprecated and may be removed in a future version. Use the `image` attribute instead.

cardimage

Deprecated since version 0.1: The `cardimage` function is deprecated and may be removed in a future version. Use the `image` attribute instead.

comment

Get the comment attribute from the card image if not already set.

field_specifier

The field-specifier of record-valued keyword cards; always `None` on normal cards.

classmethod fromstring (*image*)

Construct a `Card` object from a (raw) string. It will pad the string if it is not the length of a card image (80 columns). If the card image is longer than 80 columns, assume it contains `CONTINUE` card(s).

image

The card “image”, that is, the 80 byte character string that represents this card in an actual FITS header.

is_blank

`True` if the card is completely blank—that is, it has no keyword, value, or comment. It appears in the header as 80 spaces.

Returns `False` otherwise.

key

Deprecated since version 0.1: The `key` function is deprecated and may be removed in a future version. Use the `keyword` attribute instead.

keyword

Returns the keyword name parsed from the card image.

length = 80

The length of a `Card` image; should always be 80 for valid FITS files.

classmethod `normalize_keyword` (*keyword*)

`classmethod` to convert a keyword value that may contain a field-specifier to uppercase. The effect is to raise the key to uppercase and leave the field specifier in its original case.

Parameters

key : or str

A keyword value or a `keyword.field-specifier` value

rawkeyword

On record-valued keyword cards this is the name of the standard ≤ 8 character FITS keyword that this RVKC is stored in. Otherwise it is the card's normal keyword.

rawvalue

On record-valued keyword cards this is the raw string value in the `<field-specifier>: <value>` format stored in the card in order to represent a RVKC. Otherwise it is the card's normal value.

run_option (*option=u'warn', err_text=u'', fix_text=u'Fixed.', fix=None, fixable=True*)

Execute the verification with selected option.

value

The value associated with the keyword stored in this card.

verify (*option=u'warn'*)

Verify all values in the instance.

Parameters

option : str

Output verification option. Must be one of "fix", "silentfix", "ignore", "warn", or "exception". May also be any combination of "fix" or "silentfix" with "+ignore", +warn, or +exception" (e.g. ```fix+warn```). See [Verification options](#) for more info.

Deprecated Interfaces

The following classes and functions are deprecated as of the PyFITS 3.1 header refactoring, though they are currently still available for backwards-compatibility.

class `astropy.io.fits.CardList` (*cards=[], keylist=None*)

Bases: `list`

Deprecated since version 0.1: `CardList` used to provide the list-like functionality for manipulating a header as a list of cards. This functionality is now subsumed into the `Header` class itself, so it is no longer necessary to create or use `CardLists`.

Construct the `CardList` object from a list of `Card` objects.

`CardList` is now merely a thin wrapper around `Header` to provide backwards compatibility for the old API. This should not be used for any new code.

Parameters

cards

A list of `Card` objects.

append (**args, **kwargs*)

Deprecated since version 0.1: The `append` function is deprecated and may be removed in a future version. Use `Header.append()` instead.

Append a `Card` to the `CardList`.

Parameters**card** : `Card` objectThe `Card` to be appended.**useblanks** : bool, optionalUse any *extra* blank cards?

If `useblanks` is `True`, and if there are blank cards directly before `END`, it will use this space first, instead of appending after these blank cards, so the total space will not increase. When `useblanks` is `False`, the card will be appended at the end, even if there are blank cards in front of `END`.

bottom : bool, optional

If `False` the card will be appended after the last non-commentary card. If `True` the card will be appended after the last non-blank card.

copy (**args*, ***kwargs*)

Deprecated since version 0.1: The copy function is deprecated and may be removed in a future version. Use `Header.copy()` instead.

Make a (deep)copy of the `CardList`.

count (**args*, ***kwargs*)

Deprecated since version 0.1: The count function is deprecated and may be removed in a future version. Use `Header.count()` instead.

count_blanks (**args*, ***kwargs*)

Deprecated since version 0.1: The count_blanks function is deprecated and may be removed in a future version.

Returns how many blank cards are *directly* before the `END` card.

extend (**args*, ***kwargs*)

Deprecated since version 0.1: The extend function is deprecated and may be removed in a future version. Use `Header.extend()` instead.

filter_list (**args*, ***kwargs*)

Deprecated since version 0.1: The filter_list function is deprecated and may be removed in a future version. Use `header[<wildcard_pattern>]` instead.

Construct a `CardList` that contains references to all of the cards in this `CardList` that match the input key value including any special filter keys (`*`, `?`, and `.`).

Parameters**key** : str

key value to filter the list with

Returns

cardlist

A `CardList` object containing references to all the requested cards.

index (**args*, ***kwargs*)

Deprecated since version 0.1: The index function is deprecated and may be removed in a future version. Use `Header.index()` instead.

index_of (**args*, ***kwargs*)

Deprecated since version 0.1: The index_of function is deprecated and may be removed in a future version. Use `Header.index()` instead.

Get the index of a keyword in the `CardList`.

Parameters

key : str or int

The keyword name (a string) or the index (an integer).

backward : bool, optional

When `True`, search the index from the END, i.e., backward.

Returns

index : int

The index of the `Card` with the given keyword.

insert (*args, **kwargs)

Deprecated since version 0.1: The insert function is deprecated and may be removed in a future version. Use `Header.insert()` instead.

Insert a `Card` to the `CardList`.

Parameters

pos : int

The position (index, keyword name will not be allowed) to insert. The new card will be inserted before it.

card : `Card` object

The card to be inserted.

useblanks : bool, optional

If `useblanks` is `True`, and if there are blank cards directly before END, it will use this space first, instead of appending after these blank cards, so the total space will not increase. When `useblanks` is `False`, the card will be appended at the end, even if there are blank cards in front of END.

keys (*args, **kwargs)

Deprecated since version 0.1: The keys function is deprecated and may be removed in a future version. Use `Header.keys()` instead.

Return a list of all keywords from the `CardList`.

pop (*args, **kwargs)

Deprecated since version 0.1: The pop function is deprecated and may be removed in a future version. Use `Header.pop()` instead.

remove (*args, **kwargs)

Deprecated since version 0.1: The remove function is deprecated and may be removed in a future version. Use `Header.remove()` instead.

values (*args, **kwargs)

Deprecated since version 0.1: The values function is deprecated and may be removed in a future version. Use `Header.values()` instead.

Return a list of the values of all cards in the `CardList`.

For `RecordValuedKeywordCard` objects, the value returned is the floating point value, exclusive of the `field_specifier`.

`astropy.io.fits.create_card` (*args, **kwargs)

Deprecated since version 0.1: The `create_card` function is deprecated and may be removed in a future version. Use `Card.__init__` instead.

`astropy.io.fits.create_card_from_string(*args, **kwargs)`

Deprecated since version 0.1: The `create_card_from_string` function is deprecated and may be removed in a future version. Use `Card.fromstring()` instead.

Construct a `Card` object from a (raw) string. It will pad the string if it is not the length of a card image (80 columns). If the card image is longer than 80 columns, assume it contains CONTINUE card(s).

`astropy.io.fits.upper_key(*args, **kwargs)`

Deprecated since version 0.1: The `upper_key` function is deprecated and may be removed in a future version. Use `Card.normalize_keyword()` instead.

`classmethod` to convert a keyword value that may contain a field-specifier to uppercase. The effect is to raise the key to uppercase and leave the field specifier in its original case.

Parameters

key : or str

A keyword value or a `keyword.field-specifier` value

14.5.6 Tables

`BinTableHDU`

`class astropy.io.fits.BinTableHDU(data=None, header=None, name=None, uint=False)`

Bases: `astropy.io.fits.hdu.table._TableBaseHDU`

Binary table HDU class.

Parameters

header : Header instance

header to be used

data : array

data to be used

name : str

name to be populated in EXTNAME keyword

uint : bool, optional

set to `True` if the table contains unsigned integer columns.

`add_checksum(when=None, override_datasum=False, blocking='standard', checksum_keyword='CHECKSUM', datasum_keyword='DATASUM')`

Add the CHECKSUM and DATASUM cards to this HDU with the values set to the checksum calculated for the HDU and the data respectively. The addition of the DATASUM card may be overridden.

Parameters

when : str, optional

comment string for the cards; by default the comments will represent the time when the checksum was calculated

override_datasum : bool, optional

add the CHECKSUM card only

blocking : str, optional

“standard” or “nonstandard”, compute sum 2880 bytes at a time, or not

checksum_keyword : str, optional

The name of the header keyword to store the checksum value in; this is typically ‘CHECKSUM’ per convention, but there exist use cases in which a different keyword should be used

datasum_keyword : str, optional

See `checksum_keyword`

Notes

For testing purposes, first call `add_datasum` with a `when` argument, then call `add_checksum` with a `when` argument and `override_datasum` set to `True`. This will provide consistent comments for both cards and enable the generation of a CHECKSUM card with a consistent value.

add_datasum (*when=None, blocking='standard', datasum_keyword='DATASUM'*)

Add the DATASUM card to this HDU with the value set to the checksum calculated for the data.

Parameters

when : str, optional

Comment string for the card that by default represents the time when the checksum was calculated

blocking : str, optional

“standard” or “nonstandard”, compute sum 2880 bytes at a time, or not

datasum_keyword : str, optional

The name of the header keyword to store the datasum value in; this is typically ‘DATASUM’ per convention, but there exist use cases in which a different keyword should be used

Returns

checksum : int

The calculated datasum

Notes

For testing purposes, provide a `when` argument to enable the comment value in the card to remain consistent. This will enable the generation of a CHECKSUM card with a consistent value.

columns

The `ColDefs` objects describing the columns in this table.

copy()

Make a copy of the table HDU, both header and data are copied.

dump (*datafile=None, cdfile=None, hfile=None, clobber=False*)

Dump the table HDU to a file in ASCII format. The table may be dumped in three separate files, one containing column definitions, one containing header parameters, and one for table data.

Parameters

datafile : file path, file object or file-like object, optional

Output data file. The default is the root name of the fits file associated with this HDU appended with the extension `.txt`.

cdfile : file path, file object or file-like object, optional

Output column definitions file. The default is `None`, no column definitions output is produced.

hfile : file path, file object or file-like object, optional

Output header parameters file. The default is `None`, no header parameters output is produced.

clobber : bool

Overwrite the output files if they exist.

Notes

The primary use for the `dump` method is to allow viewing and editing the table data and parameters in a standard text editor. The `load` method can be used to create a new table from the three plain text (ASCII) files.

- datafile**: Each line of the data file represents one row of table data. The data is output one column at a time in column order. If a column contains an array, each element of the column array in the current row is output before moving on to the next column. Each row ends with a new line.

Integer data is output right-justified in a 21-character field followed by a blank. Floating point data is output right justified using 'g' format in a 21-character field with 15 digits of precision, followed by a blank. String data that does not contain whitespace is output left-justified in a field whose width matches the width specified in the `TFORM` header parameter for the column, followed by a blank. When the string data contains whitespace characters, the string is enclosed in quotation marks (""). For the last data element in a row, the trailing blank in the field is replaced by a new line character.

For column data containing variable length arrays ('P' format), the array data is preceded by the string 'VLA_Length=' and the integer length of the array for that row, left-justified in a 21-character field, followed by a blank.

Note: This format does *not* support variable length arrays using the ('Q' format) due to difficult to overcome ambiguities. What this means is that this file format cannot support VLA columns in tables stored in files that are over 2 GB in size.

For column data representing a bit field ('X' format), each bit value in the field is output right-justified in a 21-character field as 1 (for true) or 0 (for false).

- cdfile**: Each line of the column definitions file provides the definitions for one column in the table. The line is broken up into 8, sixteen-character fields. The first field provides the column name (`TTYPEn`). The second field provides the column format (`TFORMn`). The third field provides the display format (`TDISPn`). The fourth field provides the physical units (`TUNITn`). The fifth field provides the dimensions for a multidimensional array (`TDIMn`). The sixth field provides the value that signifies an undefined value (`TNULLn`). The seventh field provides the scale factor (`TSCALn`). The eighth field provides the offset value (`TZEROn`). A field value of "" is used to represent the case where no value is provided.
- hfile**: Each line of the header parameters file provides the definition of a single HDU header card as represented by the card image.

filebytes ()

Calculates and returns the number of bytes that this HDU will write to a file.

fileinfo ()

Returns a dictionary detailing information about the locations of this HDU within any associated file.

The values are only valid after a read or write of the associated file with no intervening changes to the `HDUList`.

Returns

dict or None

The dictionary details information about the locations of this HDU within an associated file. Returns `None` when the HDU is not associated with a file.

Dictionary contents:

Key	Value
<code>file</code>	File object associated with the HDU
<code>file-mode</code>	Mode in which the file was opened (readonly, copyonwrite, update, append, ostream)
<code>hdrLoc</code>	Starting byte location of header in file
<code>datLoc</code>	Starting byte location of data block in file
<code>datSpan</code>	Data size including padding

classmethod `from_columns` (*columns*, *header=None*, *nrows=0*, *fill=False*, ***kwargs*)

Given either a `ColDefs` object, a sequence of `Column` objects, or another table HDU or table data (a `FITS_rec` or multi-field `numpy.ndarray` or `numpy.recarray` object, return a new table HDU of the class this method was called on using the column definition from the input.

This is an alternative to the now deprecated `new_table` function, and otherwise accepts the same arguments. See also `FITS_rec.from_columns`.

Parameters

columns : sequence of `Column`, `ColDefs`, or other

The columns from which to create the table data, or an object with a column-like structure from which a `ColDefs` can be instantiated. This includes an existing `BinTableHDU` or `TableHDU`, or a `numpy.recarray` to give some examples.

If these columns have data arrays attached that data may be used in initializing the new table. Otherwise the input columns will be used as a template for a new table with the requested number of rows.

header : `Header`

An optional `Header` object to instantiate the new HDU yet. Header keywords specifically related to defining the table structure (such as the “TXXXn” keywords like `TYPEn`) will be overridden by the supplied column definitions, but all other informational and data model-specific keywords are kept.

nrows : int

Number of rows in the new table. If the input columns have data associated with them, the size of the largest input column is used. Otherwise the default is 0.

fill : bool

If `True`, will fill all cells with zeros or blanks. If `False`, copy the data from input, undefined cells will still be filled with zeros/blanks.

Notes

Any additional keyword arguments accepted by the HDU class’s `__init__` may also be passed in as keyword arguments.

classmethod fromstring (*data*, *checksum=False*, *ignore_missing_end=False*, ***kwargs*)

Creates a new HDU object of the appropriate type from a string containing the HDU's entire header and, optionally, its data.

Note: When creating a new HDU from a string without a backing file object, the data of that HDU may be read-only. It depends on whether the underlying string was an immutable Python str/bytes object, or some kind of read-write memory buffer such as a `memoryview`.

Parameters

data : str, bytearray, memoryview, ndarray

A byte string containing the HDU's header and data.

checksum : bool, optional

Check the HDU's checksum and/or datasum.

ignore_missing_end : bool, optional

Ignore a missing end card in the header data. Note that without the end card the end of the header may be ambiguous and result in a corrupt HDU. In this case the assumption is that the first 2880 block that does not begin with valid FITS header data is the beginning of the data.

kwargs : optional

May consist of additional keyword arguments specific to an HDU type—these correspond to keywords recognized by the constructors of different HDU classes such as `PrimaryHDU`, `ImageHDU`, or `BinTableHDU`. Any unrecognized keyword arguments are simply ignored.

classmethod load (*datafile*, *cdfile=None*, *hfile=None*, *replace=False*, *header=None*)

Create a table from the input ASCII files. The input is from up to three separate files, one containing column definitions, one containing header parameters, and one containing column data.

The column definition and header parameters files are not required. When absent the column definitions and/or header parameters are taken from the header object given in the header argument; otherwise sensible defaults are inferred (though this mode is not recommended).

Parameters

datafile : file path, file object or file-like object

Input data file containing the table data in ASCII format.

cdfile : file path, file object, file-like object, optional

Input column definition file containing the names, formats, display formats, physical units, multidimensional array dimensions, undefined values, scale factors, and offsets associated with the columns in the table. If `None`, the column definitions are taken from the current values in this object.

hfile : file path, file object, file-like object, optional

Input parameter definition file containing the header parameter definitions to be associated with the table. If `None`, the header parameter definitions are taken from the current values in this object's header.

replace : bool

When `True`, indicates that the entire header should be replaced with the contents of the ASCII file instead of just updating the current header.

header : Header object

When the `cdfile` and `hfile` are missing, use this Header object in the creation of the new table and HDU. Otherwise this Header supercedes the keywords from `hfile`, which is only used to update values not present in this Header, unless `replace=True` in which this Header's values are completely replaced with the values from `hfile`.

Notes

The primary use for the `load` method is to allow the input of ASCII data that was edited in a standard text editor of the table data and parameters. The `dump` method can be used to create the initial ASCII files.

- datafile:** Each line of the data file represents one row of table data. The data is output one column at a time in column order. If a column contains an array, each element of the column array in the current row is output before moving on to the next column. Each row ends with a new line.

Integer data is output right-justified in a 21-character field followed by a blank. Floating point data is output right justified using 'g' format in a 21-character field with 15 digits of precision, followed by a blank. String data that does not contain whitespace is output left-justified in a field whose width matches the width specified in the `TFORM` header parameter for the column, followed by a blank. When the string data contains whitespace characters, the string is enclosed in quotation marks (""). For the last data element in a row, the trailing blank in the field is replaced by a new line character.

For column data containing variable length arrays ('P' format), the array data is preceded by the string 'VLA_Length=' and the integer length of the array for that row, left-justified in a 21-character field, followed by a blank.

Note: This format does *not* support variable length arrays using the ('Q' format) due to difficult to overcome ambiguities. What this means is that this file format cannot support VLA columns in tables stored in files that are over 2 GB in size.

For column data representing a bit field ('X' format), each bit value in the field is output right-justified in a 21-character field as 1 (for true) or 0 (for false).

- cdfile:** Each line of the column definitions file provides the definitions for one column in the table. The line is broken up into 8, sixteen-character fields. The first field provides the column name (`TTYPEn`). The second field provides the column format (`TFORMn`). The third field provides the display format (`TDISPn`). The fourth field provides the physical units (`TUNITn`). The fifth field provides the dimensions for a multidimensional array (`TDIMn`). The sixth field provides the value that signifies an undefined value (`TNULLn`). The seventh field provides the scale factor (`TSCALn`). The eighth field provides the offset value (`TZERON`). A field value of "" is used to represent the case where no value is provided.

- hfile:** Each line of the header parameters file provides the definition of a single HDU header card as represented by the card image.

classmethod `readfrom` (*fileobj*, *checksum=False*, *ignore_missing_end=False*, ***kwargs*)

Read the HDU from a file. Normally an HDU should be opened with `open()` which reads the entire HDU list in a FITS file. But this method is still provided for symmetry with `writeto()`.

Parameters

fileobj : file object or file-like object

Input FITS file. The file's seek pointer is assumed to be at the beginning of the HDU.

checksum : bool

If `True`, verifies that both `DATASUM` and `CHECKSUM` card values (when present in the HDU header) match the header and data of all HDU's in the file.

ignore_missing_end : bool

Do not issue an exception when opening a file that is missing an END card in the last header.

req_cards (*keyword, pos, test, fix_value, option, errlist*)

Check the existence, location, and value of a required `Card`.

Parameters

keyword : str

The keyword to validate

pos : int, callable

If an `int`, this specifies the exact location this card should have in the header. Remember that Python is zero-indexed, so this means `pos=0` requires the card to be the first card in the header. If given a callable, it should take one argument—the actual position of the keyword—and return `True` or `False`. This can be used for custom evaluation. For example if `pos=lambda idx: idx > 10` this will check that the keyword's index is greater than 10.

test : callable

This should be a callable (generally a function) that is passed the value of the given keyword and returns `True` or `False`. This can be used to validate the value associated with the given keyword.

fix_value : str, int, float, complex, bool, None

A valid value for a FITS keyword to use if the given `test` fails to replace an invalid value. In other words, this provides a default value to use as a replacement if the keyword's current value is invalid. If `None`, there is no replacement value and the keyword is unfixable.

option : str

Output verification option. Must be one of "fix", "silentfix", "ignore", "warn", or "exception". May also be any combination of "fix" or "silentfix" with "+ignore", "+warn", or "+exception" (e.g. ```fix+warn```). See [Verification options](#) for more info.

errlist : list

A list of validation errors already found in the FITS file; this is used primarily for the validation system to collect errors across multiple HDUs and multiple calls to `req_cards`.

Notes

If `pos=None`, the card can be anywhere in the header. If the card does not exist, the new card will have the `fix_value` as its value when created. Also check the card's value by using the `test` argument.

run_option (*option=u'warn', err_text=u'', fix_text=u'Fixed.', fix=None, fixable=True*)

Execute the verification with selected option.

size

Size (in bytes) of the data portion of the HDU.

classmethod tcreate (**args, **kwargs*)

Deprecated since version 0.1: The `tcreate` method is deprecated and may be removed in a future version. Use `load()` instead.

tdump (*args, **kwargs)

Deprecated since version 0.1: The `tdump` function is deprecated and may be removed in a future version. Use `dump()` instead.

update ()

Update header keywords to reflect recent changes of columns.

update_ext_name (*args, **kwargs)

Deprecated since version 0.3: The `update_ext_name` function will be deprecated in a future version. Use the `.name` attribute or `Header.set` instead.

Update the extension name associated with the HDU.

If the keyword already exists in the Header, it's value and/or comment will be updated. If it does not exist, a new card will be created and it will be placed before or after the specified location. If no `before` or `after` is specified, it will be appended at the end.

Parameters

value : str

Value to be used for the new extension name

comment : str, optional

To be used for updating, default=None.

before : str or int, optional

Name of the keyword, or index of the `Card` before which the new card will be placed in the Header. The argument `before` takes precedence over `after` if both are specified.

after : str or int, optional

Name of the keyword, or index of the `Card` after which the new card will be placed in the Header

savecomment : bool, optional

When `True`, preserve the current comment for an existing keyword. The argument `savecomment` takes precedence over `comment` if both specified. If `comment` is not specified then the current comment will automatically be preserved.

update_ext_version (*args, **kwargs)

Deprecated since version 0.3: The `update_ext_version` function will be deprecated in a future version. Use the `.ver` attribute or `Header.set` instead.

Update the extension version associated with the HDU.

If the keyword already exists in the Header, it's value and/or comment will be updated. If it does not exist, a new card will be created and it will be placed before or after the specified location. If no `before` or `after` is specified, it will be appended at the end.

Parameters

value : str

Value to be used for the new extension version

comment : str, optional

To be used for updating; default=None.

before : str or int, optional

Name of the keyword, or index of the `Card` before which the new card will be placed in the Header. The argument `before` takes precedence over `after` if both are specified.

after : str or int, optional

Name of the keyword, or index of the `Card` after which the new card will be placed in the Header.

savecomment : bool, optional

When `True`, preserve the current comment for an existing keyword. The argument `savecomment` takes precedence over `comment` if both specified. If `comment` is not specified then the current comment will automatically be preserved.

verify (*option=u'warn'*)

Verify all values in the instance.

Parameters

option : str

Output verification option. Must be one of "fix", "silentfix", "ignore", "warn", or "exception". May also be any combination of "fix" or "silentfix" with "+ignore", "+warn", or "+exception" (e.g. ``"fix+warn"``). See *Verification options* for more info.

verify_checksum (*blocking='standard'*)

Verify that the value in the `CHECKSUM` keyword matches the value calculated for the current HDU `CHECKSUM`.

blocking: str, optional

“standard” or “nonstandard”, compute sum 2880 bytes at a time, or not

Returns

valid : int

- 0 - failure
- 1 - success
- 2 - no `CHECKSUM` keyword present

verify_datasum (*blocking='standard'*)

Verify that the value in the `DATASUM` keyword matches the value calculated for the `DATASUM` of the current HDU data.

blocking: str, optional

“standard” or “nonstandard”, compute sum 2880 bytes at a time, or not

Returns

valid : int

- 0 - failure
- 1 - success
- 2 - no `DATASUM` keyword present

writeto (*name, output_verify='exception', clobber=False, checksum=False*)

Works similarly to the normal `writeto()`, but prepends a default `PrimaryHDU` are required by extension HDUs (which cannot stand on their own).

TableHDU

class `astropy.io.fits.TableHDU` (*data=None, header=None, name=None*)

Bases: `astropy.io.fits.hdu.table._TableBaseHDU`

FITS ASCII table extension HDU class.

add_checksum (*when=None, override_datasum=False, blocking='standard', checksum_keyword='CHECKSUM', datasum_keyword='DATASUM'*)

Add the CHECKSUM and DATASUM cards to this HDU with the values set to the checksum calculated for the HDU and the data respectively. The addition of the DATASUM card may be overridden.

Parameters

when : str, optional

comment string for the cards; by default the comments will represent the time when the checksum was calculated

override_datasum : bool, optional

add the CHECKSUM card only

blocking : str, optional

“standard” or “nonstandard”, compute sum 2880 bytes at a time, or not

checksum_keyword : str, optional

The name of the header keyword to store the checksum value in; this is typically ‘CHECKSUM’ per convention, but there exist use cases in which a different keyword should be used

datasum_keyword : str, optional

See `checksum_keyword`

Notes

For testing purposes, first call `add_datasum` with a `when` argument, then call `add_checksum` with a `when` argument and `override_datasum` set to `True`. This will provide consistent comments for both cards and enable the generation of a CHECKSUM card with a consistent value.

add_datasum (*when=None, blocking='standard', datasum_keyword='DATASUM'*)

Add the DATASUM card to this HDU with the value set to the checksum calculated for the data.

Parameters

when : str, optional

Comment string for the card that by default represents the time when the checksum was calculated

blocking : str, optional

“standard” or “nonstandard”, compute sum 2880 bytes at a time, or not

datasum_keyword : str, optional

The name of the header keyword to store the datasum value in; this is typically ‘DATASUM’ per convention, but there exist use cases in which a different keyword should be used

Returns

checksum : int

The calculated datasum

Notes

For testing purposes, provide a `when` argument to enable the comment value in the card to remain consistent. This will enable the generation of a `CHECKSUM` card with a consistent value.

`columns`

The `ColDefs` objects describing the columns in this table.

`copy()`

Make a copy of the table HDU, both header and data are copied.

`filebytes()`

Calculates and returns the number of bytes that this HDU will write to a file.

`fileinfo()`

Returns a dictionary detailing information about the locations of this HDU within any associated file. The values are only valid after a read or write of the associated file with no intervening changes to the `HDUList`.

Returns

dict or None

The dictionary details information about the locations of this HDU within an associated file. Returns `None` when the HDU is not associated with a file.

Dictionary contents:

Key	Value
<code>file</code>	File object associated with the HDU
<code>file-mode</code>	Mode in which the file was opened (readonly, copyonwrite, update, append, ostream)
<code>hdrLoc</code>	Starting byte location of header in file
<code>datLoc</code>	Starting byte location of data block in file
<code>datSpan</code>	Data size including padding

`classmethod from_columns` (*columns, header=None, nrows=0, fill=False, **kwargs*)

Given either a `ColDefs` object, a sequence of `Column` objects, or another table HDU or table data (a `FITS_rec` or multi-field `numpy.ndarray` or `numpy.recarray` object, return a new table HDU of the class this method was called on using the column definition from the input.

This is an alternative to the now deprecated `new_table` function, and otherwise accepts the same arguments. See also `FITS_rec.from_columns`.

Parameters

columns : sequence of `Column`, `ColDefs`, or other

The columns from which to create the table data, or an object with a column-like structure from which a `ColDefs` can be instantiated. This includes an existing `BinTableHDU` or `TableHDU`, or a `numpy.recarray` to give some examples.

If these columns have data arrays attached that data may be used in initializing the new table. Otherwise the input columns will be used as a template for a new table with the requested number of rows.

header : `Header`

An optional `Header` object to instantiate the new HDU yet. Header keywords specifically related to defining the table structure (such as the “TXXXn” keywords like

TTYPEn) will be overridden by the supplied column definitions, but all other informational and data model-specific keywords are kept.

nrows : int

Number of rows in the new table. If the input columns have data associated with them, the size of the largest input column is used. Otherwise the default is 0.

fill : bool

If `True`, will fill all cells with zeros or blanks. If `False`, copy the data from input, undefined cells will still be filled with zeros/blanks.

Notes

Any additional keyword arguments accepted by the HDU class's `__init__` may also be passed in as keyword arguments.

classmethod `fromstring` (*data*, *checksum=False*, *ignore_missing_end=False*, ***kwargs*)

Creates a new HDU object of the appropriate type from a string containing the HDU's entire header and, optionally, its data.

Note: When creating a new HDU from a string without a backing file object, the data of that HDU may be read-only. It depends on whether the underlying string was an immutable Python str/bytes object, or some kind of read-write memory buffer such as a `memoryview`.

Parameters

data : str, bytearray, memoryview, ndarray

A byte string containing the HDU's header and data.

checksum : bool, optional

Check the HDU's checksum and/or datasum.

ignore_missing_end : bool, optional

Ignore a missing end card in the header data. Note that without the end card the end of the header may be ambiguous and result in a corrupt HDU. In this case the assumption is that the first 2880 block that does not begin with valid FITS header data is the beginning of the data.

kwargs : optional

May consist of additional keyword arguments specific to an HDU type—these correspond to keywords recognized by the constructors of different HDU classes such as `PrimaryHDU`, `ImageHDU`, or `BinTableHDU`. Any unrecognized keyword arguments are simply ignored.

classmethod `readfrom` (*fileobj*, *checksum=False*, *ignore_missing_end=False*, ***kwargs*)

Read the HDU from a file. Normally an HDU should be opened with `open()` which reads the entire HDU list in a FITS file. But this method is still provided for symmetry with `writeto()`.

Parameters

fileobj : file object or file-like object

Input FITS file. The file's seek pointer is assumed to be at the beginning of the HDU.

checksum : bool

If `True`, verifies that both DATASUM and CHECKSUM card values (when present in the HDU header) match the header and data of all HDU's in the file.

ignore_missing_end : bool

Do not issue an exception when opening a file that is missing an END card in the last header.

req_cards (*keyword, pos, test, fix_value, option, errlist*)

Check the existence, location, and value of a required `Card`.

Parameters

keyword : str

The keyword to validate

pos : int, callable

If an `int`, this specifies the exact location this card should have in the header. Remember that Python is zero-indexed, so this means `pos=0` requires the card to be the first card in the header. If given a callable, it should take one argument—the actual position of the keyword—and return `True` or `False`. This can be used for custom evaluation. For example if `pos=lambda idx: idx > 10` this will check that the keyword's index is greater than 10.

test : callable

This should be a callable (generally a function) that is passed the value of the given keyword and returns `True` or `False`. This can be used to validate the value associated with the given keyword.

fix_value : str, int, float, complex, bool, None

A valid value for a FITS keyword to use if the given `test` fails to replace an invalid value. In other words, this provides a default value to use as a replacement if the keyword's current value is invalid. If `None`, there is no replacement value and the keyword is unfixable.

option : str

Output verification option. Must be one of "fix", "silentfix", "ignore", "warn", or "exception". May also be any combination of "fix" or "silentfix" with "+ignore", "+warn", or "+exception" (e.g. ```fix+warn```). See [Verification options](#) for more info.

errlist : list

A list of validation errors already found in the FITS file; this is used primarily for the validation system to collect errors across multiple HDUs and multiple calls to `req_cards`.

Notes

If `pos=None`, the card can be anywhere in the header. If the card does not exist, the new card will have the `fix_value` as its value when created. Also check the card's value by using the `test` argument.

run_option (*option=u'warn', err_text=u'', fix_text=u'Fixed.', fix=None, fixable=True*)

Execute the verification with selected option.

size

Size (in bytes) of the data portion of the HDU.

update ()

Update header keywords to reflect recent changes of columns.

update_ext_name (*args, **kwargs)

Deprecated since version 0.3: The `update_ext_name` function will be deprecated in a future version. Use the `.name` attribute or `Header.set` instead.

Update the extension name associated with the HDU.

If the keyword already exists in the Header, it's value and/or comment will be updated. If it does not exist, a new card will be created and it will be placed before or after the specified location. If no `before` or `after` is specified, it will be appended at the end.

Parameters

value : str

Value to be used for the new extension name

comment : str, optional

To be used for updating, default=None.

before : str or int, optional

Name of the keyword, or index of the `Card` before which the new card will be placed in the Header. The argument `before` takes precedence over `after` if both are specified.

after : str or int, optional

Name of the keyword, or index of the `Card` after which the new card will be placed in the Header

savecomment : bool, optional

When `True`, preserve the current comment for an existing keyword. The argument `savecomment` takes precedence over `comment` if both specified. If `comment` is not specified then the current comment will automatically be preserved.

update_ext_version (*args, **kwargs)

Deprecated since version 0.3: The `update_ext_version` function will be deprecated in a future version. Use the `.ver` attribute or `Header.set` instead.

Update the extension version associated with the HDU.

If the keyword already exists in the Header, it's value and/or comment will be updated. If it does not exist, a new card will be created and it will be placed before or after the specified location. If no `before` or `after` is specified, it will be appended at the end.

Parameters

value : str

Value to be used for the new extension version

comment : str, optional

To be used for updating; default=None.

before : str or int, optional

Name of the keyword, or index of the `Card` before which the new card will be placed in the Header. The argument `before` takes precedence over `after` if both are specified.

after : str or int, optional

Name of the keyword, or index of the `Card` after which the new card will be placed in the Header.

savecomment : bool, optional

When `True`, preserve the current comment for an existing keyword. The argument `savecomment` takes precedence over `comment` if both specified. If `comment` is not specified then the current comment will automatically be preserved.

verify (*option=u'warn'*)

Verify all values in the instance.

Parameters

option : str

Output verification option. Must be one of "fix", "silentfix", "ignore", "warn", or "exception". May also be any combination of "fix" or "silentfix" with "+ignore", "+warn", or "+exception" (e.g. ``"fix+warn"``). See [Verification options](#) for more info.

verify_checksum (*blocking='standard'*)

Verify that the value in the CHECKSUM keyword matches the value calculated for the current HDU CHECKSUM.

blocking: str, optional

“standard” or “nonstandard”, compute sum 2880 bytes at a time, or not

Returns

valid : int

- 0 - failure
- 1 - success
- 2 - no CHECKSUM keyword present

verify_datasum (*blocking='standard'*)

Verify that the value in the DATASUM keyword matches the value calculated for the DATASUM of the current HDU data.

blocking: str, optional

“standard” or “nonstandard”, compute sum 2880 bytes at a time, or not

Returns

valid : int

- 0 - failure
- 1 - success
- 2 - no DATASUM keyword present

writeto (*name, output_verify='exception', clobber=False, checksum=False*)

Works similarly to the normal `writeto()`, but prepends a default `PrimaryHDU` are required by extension HDUs (which cannot stand on their own).

Column

class `astropy.io.fits.Column` (*name=None, format=None, unit=None, null=None, bscale=None, bzero=None, disp=None, start=None, dim=None, array=None, ascii=None*)

Bases: `object`

Class which contains the definition of one column, e.g. `ttype`, `tform`, etc. and the array containing values for the column.

Construct a `Column` by specifying attributes. All attributes except `format` can be optional.

Parameters

name : str, optional

column name, corresponding to `TTYPE` keyword

format : str, optional

column format, corresponding to `TFORM` keyword

unit : str, optional

column unit, corresponding to `TUNIT` keyword

null : str, optional

null value, corresponding to `TNULL` keyword

bscale : int-like, optional

bscale value, corresponding to `TSCAL` keyword

bzero : int-like, optional

bzero value, corresponding to `TZERO` keyword

disp : str, optional

display format, corresponding to `TDISP` keyword

start : int, optional

column starting position (ASCII table only), corresponding to `TBCOL` keyword

dim : str, optional

column dimension corresponding to `TDIM` keyword

array : iterable, optional

a `list`, `numpy.ndarray` (or other iterable that can be used to initialize an `ndarray`) providing initial data for this column. The array will be automatically converted, if possible, to the data format of the column. In the case where non-trivial `bscale` and/or `bzero` arguments are given, the values in the array must be the *physical* values—that is, the values of column as if the scaling has already been applied (the array stored on the column object will then be converted back to its storage values).

ascii : bool, optional

set `True` if this describes a column for an ASCII table; this may be required to disambiguate the column format

copy()

Return a copy of this `Column`.

ColDefs

class `astropy.io.fits.ColDefs` (*input*, *totype=None*, *ascii=False*)

Bases: `object`

Column definitions class.

It has attributes corresponding to the `Column` attributes (e.g. `ColDefs` has the attribute `names` while `Column` has `name`). Each attribute in `ColDefs` is a list of corresponding attribute values from all `Column` objects.

Parameters**input** : sequence of `Column`, `ColDefs`, otherAn existing table HDU, an existing `ColDefs`, or any multi-field Numpy array or `numpy.recarray`.****(~~Deprecated~~) tdtype**** : str, optionalwhich table HDU, "BinTableHDU" (default) or "TableHDU" (text table). Now `ColDefs` for a normal (binary) table by default, but converted automatically to ASCII table `ColDefs` in the appropriate contexts (namely, when creating an ASCII table).**ascii** : bool**add_col** (*column*)Append one `Column` to the column definition.**change_attrib** (*col_name*, *attrib*, *new_value*)Change an attribute (in the `KEYWORD_ATTRIBUTES` list) of a `Column`.**Parameters****col_name** : str or int

The column name or index to change

attrib : str

The attribute name

new_value : object

The new value for the attribute

change_name (*col_name*, *new_name*)Change a `Column`'s name.**Parameters****col_name** : str

The current name of the column

new_name : str

The new name of the column

change_unit (*col_name*, *new_unit*)Change a `Column`'s unit.**Parameters****col_name** : str or int

The column name or index

new_unit : str

The new unit for the column

del_col (*col_name*)Delete (the definition of) one `Column`.**col_name**

[str or int] The column's name or index

info (*attrib='all'*, *output=None*)

Get attribute(s) information of the column definition.

Parameters**attrib** : str

Can be one or more of the attributes listed in `astropy.io.fits.column.KEYWORD_ATTRIBUTES`. The default is "all" which will print out all attributes. It forgives plurals and blanks. If there are two or more attribute names, they must be separated by comma(s).

output : file, optional

File-like object to output to. Outputs to stdout by default. If `False`, returns the attributes as a `dict` instead.

Notes

This function doesn't return anything by default; it just prints to stdout.

FITS_rec**class** `astropy.io.fits.FITS_rec`Bases: `numpy.core.records.recarray`

FITS record array class.

`FITS_rec` is the data part of a table HDU's data part. This is a layer over the `recarray`, so we can deal with scaled columns.

It inherits all of the standard methods from `numpy.ndarray`.

columns

A user-visible accessor for the coldefs.

See <https://aeon.stsci.edu/ssb/trac/pyfits/ticket/44>

copy (*order='C'*)

The Numpy documentation lies; `numpy.ndarray.copy` is not equivalent to `numpy.copy`. Differences include that it re-views the copied array as self's ndarray subclass, as though it were taking a slice; this means `__array_finalize__` is called and the copy shares all the array attributes (including `._convert!`). So we need to make a deep copy of all those attributes so that the two arrays truly do not share any data.

Note: The `order` argument is unsupported in Numpy 1.5 and will be ignored when used with that version.

field (*key*)

A view of a `Column`'s data as an array.

classmethod `from_columns` (*columns, nrows=0, fill=False*)

Given a `ColDefs` object of unknown origin, initialize a new `FITS_rec` object.

Note: This was originally part of the `new_table` function in the table module but was moved into a class method since most of its functionality always had more to do with initializing a `FITS_rec` object than anything else, and much of it also overlapped with `FITS_rec._scale_back`.

Parameters**columns** : sequence of `Column` or a `ColDefs`

The columns from which to create the table data. If these columns have data arrays attached that data may be used in initializing the new table. Otherwise the input columns will be used as a template for a new table with the requested number of rows.

nrows : int

Number of rows in the new table. If the input columns have data associated with them, the size of the largest input column is used. Otherwise the default is 0.

fill : bool

If `True`, will fill all cells with zeros or blanks. If `False`, copy the data from input, undefined cells will still be filled with zeros/blanks.

names

List of column names.

FITS_record

```
class astropy.io.fits.FITS_record(input, row=0, start=None, end=None, step=None, base=None,
                                 **kwargs)
```

Bases: `object`

FITS record class.

`FITS_record` is used to access records of the `FITS_rec` object. This will allow us to deal with scaled columns. It also handles conversion/scaling of columns in ASCII tables. The `FITS_record` class expects a `FITS_rec` object as input.

Parameters

input : array

The array to wrap.

row : int, optional

The starting logical row of the array.

start : int, optional

The starting column in the row associated with this object. Used for subsetting the columns of the `FITS_rec` object.

end : int, optional

The ending column in the row associated with this object. Used for subsetting the columns of the `FITS_rec` object.

field (*field*)

Get the field data of the record.

setfield (*field*, *value*)

Set the field data of the record.

Table Functions

`new_table()`

```
astropy.io.fits.new_table(*args, **kwargs)
```

Deprecated since version 0.4: The `new_table` function is deprecated and may be removed in a future version.

Use `BinTableHDU.from_columns()` for new BINARY tables or `TableHDU.from_columns()` for new ASCII tables instead.

Create a new table from the input column definitions.

Warning: Creating a new table using this method creates an in-memory *copy* of all the column arrays in the input. This is because if they are separate arrays they must be combined into a single contiguous array.

If the column data is already in a single contiguous array (such as an existing record array) it may be better to create a `BinTableHDU` instance directly. See the Astropy documentation for more details.

Parameters

input : sequence of `Column` or a `ColDefs`

The data to create a table from

header : `Header` instance

Header to be used to populate the non-required keywords

nrows : int

Number of rows in the new table

fill : bool

If `True`, will fill all cells with zeros or blanks. If `False`, copy the data from input, undefined cells will still be filled with zeros/blanks.

tbtype : str or type

Table type to be created (`BinTableHDU` or `TableHDU`) or the class name as a string. Currently only `BinTableHDU` and `TableHDU` (ASCII tables) are supported.

`tabledump()`

`astropy.io.fits.tabledump(filename, datafile=None, cdfile=None, hfile=None, ext=1, clobber=False)`

Dump a table HDU to a file in ASCII format. The table may be dumped in three separate files, one containing column definitions, one containing header parameters, and one for table data.

Parameters

filename : file path, file object or file-like object

Input fits file.

datafile : file path, file object or file-like object, optional

Output data file. The default is the root name of the input fits file appended with an underscore, followed by the extension number (ext), followed by the extension `.txt`.

cdfile : file path, file object or file-like object, optional

Output column definitions file. The default is `None`, no column definitions output is produced.

hfile : file path, file object or file-like object, optional

Output header parameters file. The default is `None`, no header parameters output is produced.

ext : int

The number of the extension containing the table HDU to be dumped.

clobber : bool

Overwrite the output files if they exist.

Notes

The primary use for the `tabledump` function is to allow editing in a standard text editor of the table data and parameters. The `tcreate` function can be used to reassemble the table from the three ASCII files.

- datafile**: Each line of the data file represents one row of table data. The data is output one column at a time in column order. If a column contains an array, each element of the column array in the current row is output before moving on to the next column. Each row ends with a new line.

Integer data is output right-justified in a 21-character field followed by a blank. Floating point data is output right justified using ‘g’ format in a 21-character field with 15 digits of precision, followed by a blank. String data that does not contain whitespace is output left-justified in a field whose width matches the width specified in the `TFORM` header parameter for the column, followed by a blank. When the string data contains whitespace characters, the string is enclosed in quotation marks (“”). For the last data element in a row, the trailing blank in the field is replaced by a new line character.

For column data containing variable length arrays (‘P’ format), the array data is preceded by the string ‘VLA_Length= ’ and the integer length of the array for that row, left-justified in a 21-character field, followed by a blank.

Note: This format does *not* support variable length arrays using the (‘Q’ format) due to difficult to overcome ambiguities. What this means is that this file format cannot support VLA columns in tables stored in files that are over 2 GB in size.

For column data representing a bit field (‘X’ format), each bit value in the field is output right-justified in a 21-character field as 1 (for true) or 0 (for false).

- cdfile**: Each line of the column definitions file provides the definitions for one column in the table. The line is broken up into 8, sixteen-character fields. The first field provides the column name (`TTYPEn`). The second field provides the column format (`TFORMn`). The third field provides the display format (`TDISPn`). The fourth field provides the physical units (`TUNITn`). The fifth field provides the dimensions for a multidimensional array (`TDIMn`). The sixth field provides the value that signifies an undefined value (`TNULLn`). The seventh field provides the scale factor (`TSCALn`). The eighth field provides the offset value (`TZERON`). A field value of “” is used to represent the case where no value is provided.

- hfile**: Each line of the header parameters file provides the definition of a single HDU header card as represented by the card image.

`tableload()`

`astropy.io.fits.tableload(datafile, cdfile, hfile=None)`

Create a table from the input ASCII files. The input is from up to three separate files, one containing column definitions, one containing header parameters, and one containing column data. The header parameters file is not required. When the header parameters file is absent a minimal header is constructed.

Parameters

datafile : file path, file object or file-like object

Input data file containing the table data in ASCII format.

cdfile : file path, file object or file-like object

Input column definition file containing the names, formats, display formats, physical units, multidimensional array dimensions, undefined values, scale factors, and offsets associated with the columns in the table.

hfile : file path, file object or file-like object, optional

Input parameter definition file containing the header parameter definitions to be associated with the table. If `None`, a minimal header is constructed.

Notes

The primary use for the `tableload` function is to allow the input of ASCII data that was edited in a standard text editor of the table data and parameters. The `tabledump` function can be used to create the initial ASCII files.

- datafile**: Each line of the data file represents one row of table data. The data is output one column at a time in column order. If a column contains an array, each element of the column array in the current row is output before moving on to the next column. Each row ends with a new line.

Integer data is output right-justified in a 21-character field followed by a blank. Floating point data is output right justified using ‘g’ format in a 21-character field with 15 digits of precision, followed by a blank. String data that does not contain whitespace is output left-justified in a field whose width matches the width specified in the `TFORM` header parameter for the column, followed by a blank. When the string data contains whitespace characters, the string is enclosed in quotation marks (“”). For the last data element in a row, the trailing blank in the field is replaced by a new line character.

For column data containing variable length arrays (‘P’ format), the array data is preceded by the string ‘VLA_Length= ’ and the integer length of the array for that row, left-justified in a 21-character field, followed by a blank.

Note: This format does *not* support variable length arrays using the (‘Q’ format) due to difficult to overcome ambiguities. What this means is that this file format cannot support VLA columns in tables stored in files that are over 2 GB in size.

For column data representing a bit field (‘X’ format), each bit value in the field is output right-justified in a 21-character field as 1 (for true) or 0 (for false).

- cdfile**: Each line of the column definitions file provides the definitions for one column in the table. The line is broken up into 8, sixteen-character fields. The first field provides the column name (`TTYPEn`). The second field provides the column format (`TFORMn`). The third field provides the display format (`TDISPn`). The fourth field provides the physical units (`TUNITn`). The fifth field provides the dimensions for a multidimensional array (`TDIMn`). The sixth field provides the value that signifies an undefined value (`TNULLn`). The seventh field provides the scale factor (`TSCALn`). The eighth field provides the offset value (`TZEROn`). A field value of “” is used to represent the case where no value is provided.
- hfile**: Each line of the header parameters file provides the definition of a single HDU header card as represented by the card image.

14.5.7 Images

ImageHDU

```
class astropy.io.fits.ImageHDU (data=None, header=None, name=None,
                               do_not_scale_image_data=False, uint=False, scale_back=None)
Bases: astropy.io.fits.hdu.image._ImageBaseHDU, astropy.io.fits.hdu.base.ExtensionHDU
FITS image extension HDU class.
```

Construct an image HDU.

Parameters

data : array

The data in the HDU.

header : Header instance

The header to be used (as a template). If `header` is `None`, a minimal header will be provided.

name : str, optional

The name of the HDU, will be the value of the keyword `EXTNAME`.

do_not_scale_image_data : bool, optional

If `True`, image data is not scaled using `BSCALE/BZERO` values when read.

uint : bool, optional

Interpret signed integer data where `BZERO` is the central value and `BSCALE == 1` as unsigned integer data. For example, `int16` data with `BZERO = 32768` and `BSCALE = 1` would be treated as `uint16` data.

scale_back : bool, optional

If `True`, when saving changes to a file that contained scaled image data, restore the data to the original type and reapply the original `BSCALE/BZERO` values. This could lead to loss of accuracy if scaling back to integer values after performing floating point operations on the data.

add_checksum (*when=None, override_datasum=False, blocking='standard', checksum_keyword='CHECKSUM', datasum_keyword='DATASUM'*)

Add the `CHECKSUM` and `DATASUM` cards to this HDU with the values set to the checksum calculated for the HDU and the data respectively. The addition of the `DATASUM` card may be overridden.

Parameters

when : str, optional

comment string for the cards; by default the comments will represent the time when the checksum was calculated

override_datasum : bool, optional

add the `CHECKSUM` card only

blocking : str, optional

“standard” or “nonstandard”, compute sum 2880 bytes at a time, or not

checksum_keyword : str, optional

The name of the header keyword to store the checksum value in; this is typically ‘`CHECKSUM`’ per convention, but there exist use cases in which a different keyword should be used

datasum_keyword : str, optional

See `checksum_keyword`

Notes

For testing purposes, first call `add_datasum` with a `when` argument, then call `add_checksum` with a `when` argument and `override_datasum` set to `True`. This will provide consistent comments for both cards and enable the generation of a `CHECKSUM` card with a consistent value.

add_datasum (*when=None, blocking='standard', datasum_keyword='DATASUM'*)

Add the `DATASUM` card to this HDU with the value set to the checksum calculated for the data.

Parameters

when : str, optional

Comment string for the card that by default represents the time when the checksum was calculated

blocking : str, optional

“standard” or “nonstandard”, compute sum 2880 bytes at a time, or not

datasum_keyword : str, optional

The name of the header keyword to store the datasum value in; this is typically ‘`DATASUM`’ per convention, but there exist use cases in which a different keyword should be used

Returns

checksum : int

The calculated datasum

Notes

For testing purposes, provide a `when` argument to enable the comment value in the card to remain consistent. This will enable the generation of a `CHECKSUM` card with a consistent value.

copy ()

Make a copy of the HDU, both header and data are copied.

data

Image/array data as a `ndarray`.

Please remember that the order of axes on an Numpy array are opposite of the order specified in the FITS file. For example for a 2D image the “rows” or y-axis are the first dimension, and the “columns” or x-axis are the second dimension.

If the data is scaled using the `BZERO` and `BSCALE` parameters, this attribute returns the data scaled to its physical values unless the file was opened with `do_not_scale_image_data=True`.

filebytes ()

Calculates and returns the number of bytes that this HDU will write to a file.

fileinfo ()

Returns a dictionary detailing information about the locations of this HDU within any associated file. The values are only valid after a read or write of the associated file with no intervening changes to the `HDUList`.

Returns

dict or None

The dictionary details information about the locations of this HDU within an associated file. Returns `None` when the HDU is not associated with a file.

Dictionary contents:

Key	Value
file	File object associated with the HDU
file-mode	Mode in which the file was opened (readonly, copyonwrite, update, append, ostream)
hdrLoc	Starting byte location of header in file
datLoc	Starting byte location of data block in file
datSpan	Data size including padding

classmethod fromstring (*data, checksum=False, ignore_missing_end=False, **kwargs*)

Creates a new HDU object of the appropriate type from a string containing the HDU's entire header and, optionally, its data.

Note: When creating a new HDU from a string without a backing file object, the data of that HDU may be read-only. It depends on whether the underlying string was an immutable Python str/bytes object, or some kind of read-write memory buffer such as a `memoryview`.

Parameters

data : str, bytearray, memoryview, ndarray

A byte string containing the HDU's header and data.

checksum : bool, optional

Check the HDU's checksum and/or datasum.

ignore_missing_end : bool, optional

Ignore a missing end card in the header data. Note that without the end card the end of the header may be ambiguous and result in a corrupt HDU. In this case the assumption is that the first 2880 block that does not begin with valid FITS header data is the beginning of the data.

kwargs : optional

May consist of additional keyword arguments specific to an HDU type—these correspond to keywords recognized by the constructors of different HDU classes such as `PrimaryHDU`, `ImageHDU`, or `BinTableHDU`. Any unrecognized keyword arguments are simply ignored.

classmethod readfrom (*fileobj, checksum=False, ignore_missing_end=False, **kwargs*)

Read the HDU from a file. Normally an HDU should be opened with `open()` which reads the entire HDU list in a FITS file. But this method is still provided for symmetry with `writeto()`.

Parameters

fileobj : file object or file-like object

Input FITS file. The file's seek pointer is assumed to be at the beginning of the HDU.

checksum : bool

If `True`, verifies that both `DATASUM` and `CHECKSUM` card values (when present in the HDU header) match the header and data of all HDU's in the file.

ignore_missing_end : bool

Do not issue an exception when opening a file that is missing an `END` card in the last header.

req_cards (*keyword, pos, test, fix_value, option, errlist*)

Check the existence, location, and value of a required `Card`.

Parameters**keyword** : str

The keyword to validate

pos : int, callable

If an `int`, this specifies the exact location this card should have in the header. Remember that Python is zero-indexed, so this means `pos=0` requires the card to be the first card in the header. If given a callable, it should take one argument—the actual position of the keyword—and return `True` or `False`. This can be used for custom evaluation. For example if `pos=lambda idx: idx > 10` this will check that the keyword's index is greater than 10.

test : callable

This should be a callable (generally a function) that is passed the value of the given keyword and returns `True` or `False`. This can be used to validate the value associated with the given keyword.

fix_value : str, int, float, complex, bool, None

A valid value for a FITS keyword to use if the given `test` fails to replace an invalid value. In other words, this provides a default value to use as a replacement if the keyword's current value is invalid. If `None`, there is no replacement value and the keyword is unfixable.

option : str

Output verification option. Must be one of "fix", "silentfix", "ignore", "warn", or "exception". May also be any combination of "fix" or "silentfix" with "+ignore", "+warn", or "+exception" (e.g. `fix+warn`). See [Verification options](#) for more info.

errlist : list

A list of validation errors already found in the FITS file; this is used primarily for the validation system to collect errors across multiple HDUs and multiple calls to `req_cards`.

Notes

If `pos=None`, the card can be anywhere in the header. If the card does not exist, the new card will have the `fix_value` as its value when created. Also check the card's value by using the `test` argument.

run_option (*option=u'warn', err_text=u'', fix_text=u'Fixed.', fix=None, fixable=True*)

Execute the verification with selected option.

scale (*type=None, option='old', bscale=1, bzero=0*)Scale image data by using `BSCALE/BZERO`.

Call to this method will scale `data` and update the keywords of `BSCALE` and `BZERO` in the HDU's header. This method should only be used right before writing to the output file, as the data will be scaled and is therefore not very usable after the call.

Parameters**type** : str, optional

destination data type, use a string representing a numpy dtype name, (e.g. `'uint8'`, `'int16'`, `'float32'` etc.). If is `None`, use the current data type.

option : str

How to scale the data: if "old", use the original BSCALE and BZERO values when the data was read/created. If "minmax", use the minimum and maximum of the data to scale. The option will be overwritten by any user specified bscale/bzero values.

bscale, bzero : int, optional

User-specified BSCALE and BZERO values

section

Access a section of the image array without loading the entire array into memory. The `Section` object returned by this attribute is not meant to be used directly by itself. Rather, slices of the section return the appropriate slice of the data, and loads *only* that section into memory.

Sections are mostly obsoleted by memmap support, but should still be used to deal with very large scaled images. See the *Data Sections* section of the PyFITS documentation for more details.

shape

Shape of the image array—should be equivalent to `self.data.shape`.

size

Size (in bytes) of the data portion of the HDU.

update_ext_name (*args, **kwargs)

Deprecated since version 0.3: The `update_ext_name` function will be deprecated in a future version. Use the `.name` attribute or `Header.set` instead.

Update the extension name associated with the HDU.

If the keyword already exists in the Header, it's value and/or comment will be updated. If it does not exist, a new card will be created and it will be placed before or after the specified location. If no `before` or `after` is specified, it will be appended at the end.

Parameters

value : str

Value to be used for the new extension name

comment : str, optional

To be used for updating, default=None.

before : str or int, optional

Name of the keyword, or index of the `Card` before which the new card will be placed in the Header. The argument `before` takes precedence over `after` if both are specified.

after : str or int, optional

Name of the keyword, or index of the `Card` after which the new card will be placed in the Header

savecomment : bool, optional

When `True`, preserve the current comment for an existing keyword. The argument `savecomment` takes precedence over `comment` if both specified. If `comment` is not specified then the current comment will automatically be preserved.

update_ext_version (*args, **kwargs)

Deprecated since version 0.3: The `update_ext_version` function will be deprecated in a future version. Use the `.ver` attribute or `Header.set` instead.

Update the extension version associated with the HDU.

If the keyword already exists in the Header, it's value and/or comment will be updated. If it does not exist, a new card will be created and it will be placed before or after the specified location. If no `before` or `after` is specified, it will be appended at the end.

Parameters

value : str

Value to be used for the new extension version

comment : str, optional

To be used for updating; default=None.

before : str or int, optional

Name of the keyword, or index of the `Card` before which the new card will be placed in the Header. The argument `before` takes precedence over `after` if both are specified.

after : str or int, optional

Name of the keyword, or index of the `Card` after which the new card will be placed in the Header.

savecomment : bool, optional

When `True`, preserve the current comment for an existing keyword. The argument `savecomment` takes precedence over `comment` if both specified. If `comment` is not specified then the current comment will automatically be preserved.

update_header ()

Update the header keywords to agree with the data.

verify (*option=u'warn'*)

Verify all values in the instance.

Parameters

option : str

Output verification option. Must be one of "fix", "silentfix", "ignore", "warn", or "exception". May also be any combination of "fix" or "silentfix" with "+ignore", "+warn", or "+exception" (e.g. `"fix+warn"`). See [Verification options](#) for more info.

verify_checksum (*blocking='standard'*)

Verify that the value in the `CHECKSUM` keyword matches the value calculated for the current HDU `CHECKSUM`.

blocking: str, optional

“standard” or “nonstandard”, compute sum 2880 bytes at a time, or not

Returns

valid : int

- 0 - failure
- 1 - success
- 2 - no `CHECKSUM` keyword present

verify_datasum (*blocking='standard'*)

Verify that the value in the `DATASUM` keyword matches the value calculated for the `DATASUM` of the current HDU data.

blocking: str, optional

“standard” or “nonstandard”, compute sum 2880 bytes at a time, or not

Returns

valid : int

- 0 - failure
- 1 - success
- 2 - no DATASUM keyword present

writeto (*name*, *output_verify*='exception', *clobber*=False, *checksum*=False)

Works similarly to the normal writeto(), but prepends a default `PrimaryHDU` are required by extension HDUs (which cannot stand on their own).

CompImageHDU

```
class astropy.io.fits.CompImageHDU (data=None, header=None, name=None, compression_type='RICE_1', tile_size=None, hcomp_scale=0, hcomp_smooth=0, quantize_level=16.0, quantize_method=-1, dither_seed=0, do_not_scale_image_data=False, uint=False, scale_back=False, **kwargs)
```

Bases: `astropy.io.fits.BinTableHDU`

Compressed Image HDU class.

Parameters

data : array, optional

Uncompressed image data

header : Header instance, optional

Header to be associated with the image; when reading the HDU from a file (data=DELAYED), the header read from the file

name : str, optional

The EXTNAME value; if this value is `None`, then the name from the input image header will be used; if there is no name in the input image header then the default name COMPRESSED_IMAGE is used.

compression_type : str, optional

Compression algorithm: one of 'RICE_1', 'RICE_ONE', 'PLIO_1', 'GZIP_1', 'GZIP_2', 'HCOMPRESS_1'

tile_size : int, optional

Compression tile sizes. Default treats each row of image as a tile.

hcomp_scale : float, optional

HCOMPRESS scale parameter

hcomp_smooth : float, optional

HCOMPRESS smooth parameter

quantize_level : float, optional

Floating point quantization level; see note below

quantize_method : int, optional

Floating point quantization dithering method; can be either `NO_DITHER` (-1), `SUBTRACTIVE_DITHER_1` (1; default), or `SUBTRACTIVE_DITHER_2` (2); see note below

dither_seed : int, optional

Random seed to use for dithering; can be either an integer in the range 1 to 1000 (inclusive), `DITHER_SEED_CLOCK` (0; default), or `DITHER_SEED_CHECKSUM` (-1); see note below

Notes

The `astropy.io.fits` package supports 2 methods of image compression:

1. The entire FITS file may be externally compressed with the `gzip` or `pkzip` utility programs, producing a `*.gz` or `*.zip` file, respectively. When reading compressed files of this type, Astropy first uncompresses the entire file into a temporary file before performing the requested read operations. The `astropy.io.fits` package does not support writing to these types of compressed files. This type of compression is supported in the `_File` class, not in the `CompImageHDU` class. The file compression type is recognized by the `.gz` or `.zip` file name extension.
2. The `CompImageHDU` class supports the FITS tiled image compression convention in which the image is subdivided into a grid of rectangular tiles, and each tile of pixels is individually compressed. The details of this FITS compression convention are described at the [FITS Support Office web site](#). Basically, the compressed image tiles are stored in rows of a variable length array column in a FITS binary table. The `astropy.io.fits` recognizes that this binary table extension contains an image and treats it as if it were an image extension. Under this tile-compression format, FITS header keywords remain uncompressed. At this time, Astropy does not support the ability to extract and uncompress sections of the image without having to uncompress the entire image.

The `astropy.io.fits` package supports 3 general-purpose compression algorithms plus one other special-purpose compression technique that is designed for data masks with positive integer pixel values. The 3 general purpose algorithms are GZIP, Rice, and HCOMPRESS, and the special-purpose technique is the IRAF pixel list compression technique (PLIO). The `compression_type` parameter defines the compression algorithm to be used.

The FITS image can be subdivided into any desired rectangular grid of compression tiles. With the GZIP, Rice, and PLIO algorithms, the default is to take each row of the image as a tile. The HCOMPRESS algorithm is inherently 2-dimensional in nature, so the default in this case is to take 16 rows of the image per tile. In most cases, it makes little difference what tiling pattern is used, so the default tiles are usually adequate. In the case of very small images, it could be more efficient to compress the whole image as a single tile. Note that the image dimensions are not required to be an integer multiple of the tile dimensions; if not, then the tiles at the edges of the image will be smaller than the other tiles. The `tile_size` parameter may be provided as a list of tile sizes, one for each dimension in the image. For example a `tile_size` value of `[100, 100]` would divide a 300 X 300 image into 9 100 X 100 tiles.

The 4 supported image compression algorithms are all ‘lossless’ when applied to integer FITS images; the pixel values are preserved exactly with no loss of information during the compression and uncompression process. In addition, the HCOMPRESS algorithm supports a ‘lossy’ compression mode that will produce larger amount of image compression. This is achieved by specifying a non-zero value for the `hcomp_scale` parameter. Since the amount of compression that is achieved depends directly on the RMS noise in the image, it is usually more convenient to specify the `hcomp_scale` factor relative to the RMS noise. Setting `hcomp_scale = 2.5` means use a scale factor that is 2.5 times the calculated RMS noise in the image tile. In some cases it may be desirable to specify the exact scaling to be used, instead of specifying it relative to the calculated noise value. This may be done by specifying the negative of the desired scale value (typically in the range -2 to -100).

Very high compression factors (of 100 or more) can be achieved by using large `hcomp_scale` values, however, this can produce undesirable ‘blocky’ artifacts in the compressed image. A variation of the HCOMPRESS algorithm (called HSCOMPRESS) can be used in this case to apply a small amount of smoothing of the image when it is uncompressed to help cover up these artifacts. This smoothing is purely cosmetic and does not cause any significant change to the image pixel values. Setting the `hcomp_smooth` parameter to 1 will engage the smoothing algorithm.

Floating point FITS images (which have `BITPIX = -32` or `-64`) usually contain too much ‘noise’ in the least significant bits of the mantissa of the pixel values to be effectively compressed with any lossless algorithm. Consequently, floating point images are first quantized into scaled integer pixel values (and thus throwing away much of the noise) before being compressed with the specified algorithm (either GZIP, RICE, or HCOMPRESS). This technique produces much higher compression factors than simply using the GZIP utility to externally compress the whole FITS file, but it also means that the original floating point value pixel values are not exactly preserved. When done properly, this integer scaling technique will only discard the insignificant noise while still preserving all the real information in the image. The amount of precision that is retained in the pixel values is controlled by the `quantize_level` parameter. Larger values will result in compressed images whose pixels more closely match the floating point pixel values, but at the same time the amount of compression that is achieved will be reduced. Users should experiment with different values for this parameter to determine the optimal value that preserves all the useful information in the image, without needlessly preserving all the ‘noise’ which will hurt the compression efficiency.

The default value for the `quantize_level` scale factor is 16, which means that scaled integer pixel values will be quantized such that the difference between adjacent integer values will be 1/16th of the noise level in the image background. An optimized algorithm is used to accurately estimate the noise in the image. As an example, if the RMS noise in the background pixels of an image = 32.0, then the spacing between adjacent scaled integer pixel values will equal 2.0 by default. Note that the RMS noise is independently calculated for each tile of the image, so the resulting integer scaling factor may fluctuate slightly for each tile. In some cases, it may be desirable to specify the exact quantization level to be used, instead of specifying it relative to the calculated noise value. This may be done by specifying the negative of desired quantization level for the value of `quantize_level`. In the previous example, one could specify `quantize_level = -2.0` so that the quantized integer levels differ by 2.0. Larger negative values for `quantize_level` means that the levels are more coarsely-spaced, and will produce higher compression factors.

The quantization algorithm can also apply one of two random dithering methods in order to reduce bias in the measured intensity of background regions. The default method, specified with the constant `SUBTRACTIVE_DITHER_1` adds dithering to the zero-point of the quantization array itself rather than adding noise to the actual image. The random noise is added on a pixel-by-pixel basis, so in order restore each pixel from its integer value to its floating point value it is necessary to replay the same sequence of random numbers for each pixel (see below). The other method, `SUBTRACTIVE_DITHER_2`, is exactly like the first except that before dithering any pixel with a floating point value of `0.0` is replaced with the special integer value `-2147483647`. When the image is uncompressed, pixels with this value are restored back to `0.0` exactly. Finally, a value of `NO_DITHER` disables dithering entirely.

As mentioned above, when using the subtractive dithering algorithm it is necessary to be able to generate a (pseudo-)random sequence of noise for each pixel, and replay that same sequence upon decompressing. To facilitate this, a random seed between 1 and 10000 (inclusive) is used to seed a random number generator, and that seed is stored in the `ZDITHER0` keyword in the header of the compressed HDU. In order to use that seed to generate the same sequence of random numbers the same random number generator must be used at compression and decompression time; for that reason the tiled image convention provides an implementation of a very simple pseudo-random number generator. The seed itself can be provided in one of three ways, controllable by the `dither_seed` argument: It may be specified manually, or it may be generated arbitrarily based on the system’s clock (`DITHER_SEED_CLOCK`) or based on a checksum of the pixels in the image’s first tile (`DITHER_SEED_CHECKSUM`). The clock-based method is the default, and is sufficient to ensure that the value is reasonably “arbitrary” and that the same seed is unlikely to be generated sequentially. The checksum method, on the other hand, ensures that the same seed is used every time for a specific image. This is particularly useful for software testing as it ensures that the same image will always use the same seed.

add_checksum (*when=None*, *override_datasum=False*, *blocking='standard'*, *checksum_keyword='CHECKSUM'*, *datasum_keyword='DATASUM'*)

Add the CHECKSUM and DATASUM cards to this HDU with the values set to the checksum calculated for the HDU and the data respectively. The addition of the DATASUM card may be overridden.

Parameters

when : str, optional

comment string for the cards; by default the comments will represent the time when the checksum was calculated

override_datasum : bool, optional

add the CHECKSUM card only

blocking : str, optional

“standard” or “nonstandard”, compute sum 2880 bytes at a time, or not

checksum_keyword : str, optional

The name of the header keyword to store the checksum value in; this is typically ‘CHECKSUM’ per convention, but there exist use cases in which a different keyword should be used

datasum_keyword : str, optional

See `checksum_keyword`

Notes

For testing purposes, first call `add_datasum` with a `when` argument, then call `add_checksum` with a `when` argument and `override_datasum` set to `True`. This will provide consistent comments for both cards and enable the generation of a CHECKSUM card with a consistent value.

add_datasum (*when=None*, *blocking='standard'*, *datasum_keyword='DATASUM'*)

Add the DATASUM card to this HDU with the value set to the checksum calculated for the data.

Parameters

when : str, optional

Comment string for the card that by default represents the time when the checksum was calculated

blocking : str, optional

“standard” or “nonstandard”, compute sum 2880 bytes at a time, or not

datasum_keyword : str, optional

The name of the header keyword to store the datasum value in; this is typically ‘DATASUM’ per convention, but there exist use cases in which a different keyword should be used

Returns

checksum : int

The calculated datasum

Notes

For testing purposes, provide a `when` argument to enable the comment value in the card to remain consistent. This will enable the generation of a `CHECKSUM` card with a consistent value.

columns

The `ColDefs` objects describing the columns in this table.

compData

Deprecated since version 0.3: The `compData` function will be deprecated in a future version. Use the `compressed_data` attribute instead.

copy()

Make a copy of the table HDU, both header and data are copied.

dump (*datafile=None, cdfile=None, hfile=None, clobber=False*)

Dump the table HDU to a file in ASCII format. The table may be dumped in three separate files, one containing column definitions, one containing header parameters, and one for table data.

Parameters

datafile : file path, file object or file-like object, optional

Output data file. The default is the root name of the fits file associated with this HDU appended with the extension `.txt`.

cdfile : file path, file object or file-like object, optional

Output column definitions file. The default is `None`, no column definitions output is produced.

hfile : file path, file object or file-like object, optional

Output header parameters file. The default is `None`, no header parameters output is produced.

clobber : bool

Overwrite the output files if they exist.

Notes

The primary use for the `dump` method is to allow viewing and editing the table data and parameters in a standard text editor. The `load` method can be used to create a new table from the three plain text (ASCII) files.

•**datafile**: Each line of the data file represents one row of table data. The data is output one column at a time in column order. If a column contains an array, each element of the column array in the current row is output before moving on to the next column. Each row ends with a new line.

Integer data is output right-justified in a 21-character field followed by a blank. Floating point data is output right justified using ‘g’ format in a 21-character field with 15 digits of precision, followed by a blank. String data that does not contain whitespace is output left-justified in a field whose width matches the width specified in the `TFORM` header parameter for the column, followed by a blank. When the string data contains whitespace characters, the string is enclosed in quotation marks (“”). For the last data element in a row, the trailing blank in the field is replaced by a new line character.

For column data containing variable length arrays (‘P’ format), the array data is preceded by the string `'VLA_Length= '` and the integer length of the array for that row, left-justified in a 21-character field, followed by a blank.

Note: This format does *not* support variable length arrays using the ('Q' format) due to difficult to overcome ambiguities. What this means is that this file format cannot support VLA columns in tables stored in files that are over 2 GB in size.

For column data representing a bit field ('X' format), each bit value in the field is output right-justified in a 21-character field as 1 (for true) or 0 (for false).

- cdfile:** Each line of the column definitions file provides the definitions for one column in the table. The line is broken up into 8, sixteen-character fields. The first field provides the column name (TTYPE_n). The second field provides the column format (TFORM_n). The third field provides the display format (TDISP_n). The fourth field provides the physical units (TUNIT_n). The fifth field provides the dimensions for a multidimensional array (TDIM_n). The sixth field provides the value that signifies an undefined value (TNULL_n). The seventh field provides the scale factor (TSCALE_n). The eighth field provides the offset value (TZERO_n). A field value of " " is used to represent the case where no value is provided.
- hfile:** Each line of the header parameters file provides the definition of a single HDU header card as represented by the card image.

filebytes ()

Calculates and returns the number of bytes that this HDU will write to a file.

fileinfo ()

Returns a dictionary detailing information about the locations of this HDU within any associated file. The values are only valid after a read or write of the associated file with no intervening changes to the HDUList.

Returns

dict or None

The dictionary details information about the locations of this HDU within an associated file. Returns `None` when the HDU is not associated with a file.

Dictionary contents:

Key	Value
file	File object associated with the HDU
file-mode	Mode in which the file was opened (readonly, copyonwrite, update, append, ostream)
hdrLoc	Starting byte location of header in file
datLoc	Starting byte location of data block in file
datSpan	Data size including padding

classmethod from_columns (*columns, header=None, nrows=0, fill=False, **kwargs*)

Given either a `ColDefs` object, a sequence of `Column` objects, or another table HDU or table data (a `FITS_rec` or multi-field `numpy.ndarray` or `numpy.recarray` object, return a new table HDU of the class this method was called on using the column definition from the input.

This is an alternative to the now deprecated `new_table` function, and otherwise accepts the same arguments. See also `FITS_rec.from_columns`.

Parameters

columns : sequence of `Column`, `ColDefs`, or other

The columns from which to create the table data, or an object with a column-like structure from which a `ColDefs` can be instantiated. This includes an existing `BinTableHDU` or `TableHDU`, or a `numpy.recarray` to give some examples.

If these columns have data arrays attached that data may be used in initializing the new table. Otherwise the input columns will be used as a template for a new table with the

requested number of rows.

header : `Header`

An optional `Header` object to instantiate the new HDU yet. Header keywords specifically related to defining the table structure (such as the “TXXXn” keywords like `TTYPEn`) will be overridden by the supplied column definitions, but all other informational and data model-specific keywords are kept.

nrows : `int`

Number of rows in the new table. If the input columns have data associated with them, the size of the largest input column is used. Otherwise the default is 0.

fill : `bool`

If `True`, will fill all cells with zeros or blanks. If `False`, copy the data from input, undefined cells will still be filled with zeros/blanks.

Notes

Any additional keyword arguments accepted by the HDU class’s `__init__` may also be passed in as keyword arguments.

classmethod `fromstring` (*data*, *checksum=False*, *ignore_missing_end=False*, ***kwargs*)

Creates a new HDU object of the appropriate type from a string containing the HDU’s entire header and, optionally, its data.

Note: When creating a new HDU from a string without a backing file object, the data of that HDU may be read-only. It depends on whether the underlying string was an immutable Python `str/bytes` object, or some kind of read-write memory buffer such as a `memoryview`.

Parameters

data : `str`, `bytearray`, `memoryview`, `ndarray`

A byte string containing the HDU’s header and data.

checksum : `bool`, optional

Check the HDU’s checksum and/or datasum.

ignore_missing_end : `bool`, optional

Ignore a missing end card in the header data. Note that without the end card the end of the header may be ambiguous and resulted in a corrupt HDU. In this case the assumption is that the first 2880 block that does not begin with valid FITS header data is the beginning of the data.

kwargs : optional

May consist of additional keyword arguments specific to an HDU type—these correspond to keywords recognized by the constructors of different HDU classes such as `PrimaryHDU`, `ImageHDU`, or `BinTableHDU`. Any unrecognized keyword arguments are simply ignored.

classmethod `load` (*datafile*, *cdfile=None*, *hfile=None*, *replace=False*, *header=None*)

Create a table from the input ASCII files. The input is from up to three separate files, one containing column definitions, one containing header parameters, and one containing column data.

The column definition and header parameters files are not required. When absent the column definitions and/or header parameters are taken from the header object given in the header argument; otherwise sensible defaults are inferred (though this mode is not recommended).

Parameters**datafile** : file path, file object or file-like object

Input data file containing the table data in ASCII format.

cdfile : file path, file object, file-like object, optionalInput column definition file containing the names, formats, display formats, physical units, multidimensional array dimensions, undefined values, scale factors, and offsets associated with the columns in the table. If `None`, the column definitions are taken from the current values in this object.**hfile** : file path, file object, file-like object, optionalInput parameter definition file containing the header parameter definitions to be associated with the table. If `None`, the header parameter definitions are taken from the current values in this objects header.**replace** : boolWhen `True`, indicates that the entire header should be replaced with the contents of the ASCII file instead of just updating the current header.**header** : Header objectWhen the `cdfile` and `hfile` are missing, use this Header object in the creation of the new table and HDU. Otherwise this Header supercedes the keywords from `hfile`, which is only used to update values not present in this Header, unless `replace=True` in which this Header's values are completely replaced with the values from `hfile`.**Notes**

The primary use for the `load` method is to allow the input of ASCII data that was edited in a standard text editor of the table data and parameters. The `dump` method can be used to create the initial ASCII files.

- datafile:** Each line of the data file represents one row of table data. The data is output one column at a time in column order. If a column contains an array, each element of the column array in the current row is output before moving on to the next column. Each row ends with a new line.

Integer data is output right-justified in a 21-character field followed by a blank. Floating point data is output right justified using 'g' format in a 21-character field with 15 digits of precision, followed by a blank. String data that does not contain whitespace is output left-justified in a field whose width matches the width specified in the `TFORM` header parameter for the column, followed by a blank. When the string data contains whitespace characters, the string is enclosed in quotation marks (""). For the last data element in a row, the trailing blank in the field is replaced by a new line character.

For column data containing variable length arrays ('P' format), the array data is preceded by the string 'VLA_Length=' and the integer length of the array for that row, left-justified in a 21-character field, followed by a blank.

Note: This format does *not* support variable length arrays using the ('Q' format) due to difficult to overcome ambiguities. What this means is that this file format cannot support VLA columns in tables stored in files that are over 2 GB in size.

For column data representing a bit field ('X' format), each bit value in the field is output right-justified in a 21-character field as 1 (for true) or 0 (for false).

- cdfile:** Each line of the column definitions file provides the definitions for one column in the table. The line is broken up into 8, sixteen-character fields. The first field provides the column name (`TTYPEn`).

The second field provides the column format (TFORMn). The third field provides the display format (TDISPn). The fourth field provides the physical units (TUNITn). The fifth field provides the dimensions for a multidimensional array (TDIMn). The sixth field provides the value that signifies an undefined value (TNULLn). The seventh field provides the scale factor (TSCALn). The eighth field provides the offset value (TZEROn). A field value of " " is used to represent the case where no value is provided.

•**hfile:** Each line of the header parameters file provides the definition of a single HDU header card as represented by the card image.

classmethod readfrom (*fileobj*, *checksum=False*, *ignore_missing_end=False*, ***kwargs*)

Read the HDU from a file. Normally an HDU should be opened with `open()` which reads the entire HDU list in a FITS file. But this method is still provided for symmetry with `writeto()`.

Parameters

fileobj : file object or file-like object

Input FITS file. The file's seek pointer is assumed to be at the beginning of the HDU.

checksum : bool

If `True`, verifies that both DATASUM and CHECKSUM card values (when present in the HDU header) match the header and data of all HDU's in the file.

ignore_missing_end : bool

Do not issue an exception when opening a file that is missing an END card in the last header.

req_cards (*keyword*, *pos*, *test*, *fix_value*, *option*, *errlist*)

Check the existence, location, and value of a required `Card`.

Parameters

keyword : str

The keyword to validate

pos : int, callable

If an `int`, this specifies the exact location this card should have in the header. Remember that Python is zero-indexed, so this means `pos=0` requires the card to be the first card in the header. If given a callable, it should take one argument—the actual position of the keyword—and return `True` or `False`. This can be used for custom evaluation. For example if `pos=lambda idx: idx > 10` this will check that the keyword's index is greater than 10.

test : callable

This should be a callable (generally a function) that is passed the value of the given keyword and returns `True` or `False`. This can be used to validate the value associated with the given keyword.

fix_value : str, int, float, complex, bool, None

A valid value for a FITS keyword to use if the given `test` fails to replace an invalid value. In other words, this provides a default value to use as a replacement if the keyword's current value is invalid. If `None`, there is no replacement value and the keyword is unfixable.

option : str

Output verification option. Must be one of "fix", "silentfix", "ignore", "warn", or "exception". May also be any combination of "fix"

or "silentfix" with "+ignore", +warn, or +exception" (e.g. ``"fix+warn"). See *Verification options* for more info.

errlist : list

A list of validation errors already found in the FITS file; this is used primarily for the validation system to collect errors across multiple HDUs and multiple calls to `req_cards`.

Notes

If `pos=None`, the card can be anywhere in the header. If the card does not exist, the new card will have the `fix_value` as its value when created. Also check the card's value by using the `test` argument.

run_option (*option=u'warn', err_text=u'', fix_text=u'Fixed.', fix=None, fixable=True*)

Execute the verification with selected option.

scale (*type=None, option='old', bscale=1, bzero=0*)

Scale image data by using `BSCALE` and `BZERO`.

Calling this method will scale `self.data` and update the keywords of `BSCALE` and `BZERO` in `self._header` and `self._image_header`. This method should only be used right before writing to the output file, as the data will be scaled and is therefore not very usable after the call.

Parameters

type : str, optional

destination data type, use a string representing a numpy dtype name, (e.g. 'uint8', 'int16', 'float32' etc.). If is `None`, use the current data type.

option : str, optional

how to scale the data: if "old", use the original `BSCALE` and `BZERO` values when the data was read/created. If "minmax", use the minimum and maximum of the data to scale. The option will be overwritten by any user-specified `bscale/bzero` values.

bscale, bzero : int, optional

user specified `BSCALE` and `BZERO` values.

shape

Shape of the image array—should be equivalent to `self.data.shape`.

size

Size (in bytes) of the data portion of the HDU.

classmethod tcreate (**args, **kwargs*)

Deprecated since version 0.1: The `tcreate` method is deprecated and may be removed in a future version. Use `load()` instead.

tdump (**args, **kwargs*)

Deprecated since version 0.1: The `tdump` function is deprecated and may be removed in a future version. Use `dump()` instead.

update ()

Update header keywords to reflect recent changes of columns.

updateCompressedData (**args, **kwargs*)

Deprecated since version 0.3: The `updateCompressedData` function is deprecated and may be removed in a future version. Use (refactor your code) instead.

updateHeader (*args, **kwargs)

Deprecated since version 0.3: The updateHeader function is deprecated and may be removed in a future version. Use (refactor your code; this function no longer does anything) instead.

updateHeaderData (*args, **kwargs)

Deprecated since version 0.3: The updateHeaderData function will be deprecated in a future version. Use (refactor your code) instead.

update_ext_name (*args, **kwargs)

Deprecated since version 0.3: The update_ext_name function will be deprecated in a future version. Use the `.name` attribute or `Header.set` instead.

Update the extension name associated with the HDU.

If the keyword already exists in the Header, it's value and/or comment will be updated. If it does not exist, a new card will be created and it will be placed before or after the specified location. If no `before` or `after` is specified, it will be appended at the end.

Parameters

value : str

Value to be used for the new extension name

comment : str, optional

To be used for updating, default=None.

before : str or int, optional

Name of the keyword, or index of the `Card` before which the new card will be placed in the Header. The argument `before` takes precedence over `after` if both are specified.

after : str or int, optional

Name of the keyword, or index of the `Card` after which the new card will be placed in the Header

savecomment : bool, optional

When `True`, preserve the current comment for an existing keyword. The argument `savecomment` takes precedence over `comment` if both specified. If `comment` is not specified then the current comment will automatically be preserved.

update_ext_version (*args, **kwargs)

Deprecated since version 0.3: The update_ext_version function will be deprecated in a future version. Use the `.ver` attribute or `Header.set` instead.

Update the extension version associated with the HDU.

If the keyword already exists in the Header, it's value and/or comment will be updated. If it does not exist, a new card will be created and it will be placed before or after the specified location. If no `before` or `after` is specified, it will be appended at the end.

Parameters

value : str

Value to be used for the new extension version

comment : str, optional

To be used for updating; default=None.

before : str or int, optional

Name of the keyword, or index of the `Card` before which the new card will be placed in the Header. The argument `before` takes precedence over `after` if both are specified.

after : str or int, optional

Name of the keyword, or index of the `Card` after which the new card will be placed in the Header.

savecomment : bool, optional

When `True`, preserve the current comment for an existing keyword. The argument `savecomment` takes precedence over `comment` if both specified. If `comment` is not specified then the current comment will automatically be preserved.

verify (*option=u'warn'*)

Verify all values in the instance.

Parameters

option : str

Output verification option. Must be one of "fix", "silentfix", "ignore", "warn", or "exception". May also be any combination of "fix" or "silentfix" with "+ignore", "+warn, or "+exception" (e.g. ``"fix+warn"). See [Verification options](#) for more info.

verify_checksum (*blocking='standard'*)

Verify that the value in the `CHECKSUM` keyword matches the value calculated for the current HDU `CHECKSUM`.

blocking: str, optional

“standard” or “nonstandard”, compute sum 2880 bytes at a time, or not

Returns

valid : int

- 0 - failure
- 1 - success
- 2 - no `CHECKSUM` keyword present

verify_datasum (*blocking='standard'*)

Verify that the value in the `DATASUM` keyword matches the value calculated for the `DATASUM` of the current HDU data.

blocking: str, optional

“standard” or “nonstandard”, compute sum 2880 bytes at a time, or not

Returns

valid : int

- 0 - failure
- 1 - success
- 2 - no `DATASUM` keyword present

writeto (*name, output_verify='exception', clobber=False, checksum=False*)

Works similarly to the normal `writeto()`, but prepends a default `PrimaryHDU` are required by extension HDUs (which cannot stand on their own).

Section

class `astropy.io.fits.Section` (*hdu*)

Bases: `object`

Image section.

Slices of this object load the corresponding section of an image array from the underlying FITS file on disk, and applies any BSCALE/BZERO factors.

Section slices cannot be assigned to, and modifications to a section are not saved back to the underlying file.

See the *Data Sections* section of the PyFITS documentation for more details.

14.5.8 Differs

Facilities for diffing two FITS files. Includes objects for diffing entire FITS files, individual HDUs, FITS headers, or just FITS data.

Used to implement the fitsdiff program.

FITSDiff

class `astropy.io.fits.FITSDiff` (*a*, *b*, *ignore_keywords=[]*, *ignore_comments=[]*, *ignore_fields=[]*,
numdiffs=10, *tolerance=0.0*, *ignore_blanks=True*, *ignore_blank_cards=True*)

Bases: `astropy.io.fits.diff._BaseDiff`

Diff two FITS files by filename, or two `HDUList` objects.

`FITSDiff` objects have the following diff attributes:

- `diff_hdu_count`: If the FITS files being compared have different numbers of HDUs, this contains a 2-tuple of the number of HDUs in each file.
- `diff_hdus`: If any HDUs with the same index are different, this contains a list of 2-tuples of the HDU index and the `HDUDiff` object representing the differences between the two HDUs.

Parameters

a : str or `HDUList`

The filename of a FITS file on disk, or an `HDUList` object.

b : str or `HDUList`

The filename of a FITS file on disk, or an `HDUList` object to compare to the first file.

ignore_keywords : sequence, optional

Header keywords to ignore when comparing two headers; the presence of these keywords and their values are ignored. Wildcard strings may also be included in the list.

ignore_comments : sequence, optional

A list of header keywords whose comments should be ignored in the comparison. May contain wildcard strings as with `ignore_keywords`.

ignore_fields : sequence, optional

The (case-insensitive) names of any table columns to ignore if any table data is to be compared.

numdiffs : int, optional

The number of pixel/table values to output when reporting HDU data differences. Though the count of differences is the same either way, this allows controlling the number of different values that are kept in memory or output. If a negative value is given, then numdiffs is treated as unlimited (default: 10).

tolerance : float, optional

The relative difference to allow when comparing two float values either in header values, image arrays, or table columns (default: 0.0).

ignore_blanks : bool, optional

Ignore extra whitespace at the end of string values either in headers or data. Extra leading whitespace is not ignored (default: True).

ignore_blank_cards : bool, optional

Ignore all cards that are blank, i.e. they only contain whitespace (default: True).

classmethod fromdiff (*other, a, b*)

Returns a new Diff object of a specific subclass from an existing diff object, passing on the values for any arguments they share in common (such as ignore_keywords).

For example:

```
>>> hdul1, hdul2 = HDUList(), HDUList()
>>> headera, headerb = Header(), Header()
>>> fd = FITSDiff(hdul1, hdul2, ignore_keywords=['*'])
>>> hd = HeaderDiff.fromdiff(fd, headera, headerb)
>>> list(hd.ignore_keywords)
['*']
```

identical

True if all the `.diff_*` attributes on this diff instance are empty, implying that no differences were found.

Any subclass of `_BaseDiff` must have at least one `.diff_*` attribute, which contains a non-empty value if and only if some difference was found between the two objects being compared.

report (*fileobj=None, indent=0*)

Generates a text report on the differences (if any) between two objects, and either returns it as a string or writes it to a file-like object.

Parameters

fileobj : file-like object or None, optional

If `None`, this method returns the report as a string. Otherwise it returns `None` and writes the report to the given file-like object (which must have a `.write()` method at a minimum).

indent : int

The number of 4 space tabs to indent the report.

Returns

report : str or None

HDUdiff

```
class astropy.io.fits.HDUdiff(a, b, ignore_keywords=[], ignore_comments=[], ignore_fields=[],
                              numdiffs=10, tolerance=0.0, ignore_blanks=True, ignore_blank_cards=True)
```

Bases: `astropy.io.fits.diff._BaseDiff`

Diff two HDU objects, including their headers and their data (but only if both HDUs contain the same type of data (image, table, or unknown)).

`HDUdiff` objects have the following diff attributes:

- `diff_extnames`: If the two HDUs have different `EXTNAME` values, this contains a 2-tuple of the different extension names.
- `diff_extvers`: If the two HDUS have different `EXTVER` values, this contains a 2-tuple of the different extension versions.
- `diff_extlevels`: If the two HDUs have different `EXTLEVEL` values, this contains a 2-tuple of the different extension levels.
- `diff_extension_types`: If the two HDUs have different `XTENSION` values, this contains a 2-tuple of the different extension types.
- `diff_headers`: Contains a `HeaderDiff` object for the headers of the two HDUs. This will always contain an object—it may be determined whether the headers are different through `diff_headers.identical`.
- `diff_data`: Contains either a `ImageDataDiff`, `TableDataDiff`, or `RawDataDiff` as appropriate for the data in the HDUs, and only if the two HDUs have non-empty data of the same type (`RawDataDiff` is used for HDUs containing non-empty data of an indeterminate type).

See `FITSDiff` for explanations of the initialization parameters.

classmethod `fromdiff` (*other, a, b*)

Returns a new Diff object of a specific subclass from an existing diff object, passing on the values for any arguments they share in common (such as `ignore_keywords`).

For example:

```
>>> hdul1, hdul2 = HDUList(), HDUList()
>>> headera, headerb = Header(), Header()
>>> fd = FITSDiff(hdul1, hdul2, ignore_keywords=['*'])
>>> hd = HeaderDiff.fromdiff(fd, headera, headerb)
>>> list(hd.ignore_keywords)
['*']
```

identical

`True` if all the `.diff_*` attributes on this diff instance are empty, implying that no differences were found.

Any subclass of `_BaseDiff` must have at least one `.diff_*` attribute, which contains a non-empty value if and only if some difference was found between the two objects being compared.

report (*fileobj=None, indent=0*)

Generates a text report on the differences (if any) between two objects, and either returns it as a string or writes it to a file-like object.

Parameters

fileobj : file-like object or `None`, optional

If `None`, this method returns the report as a string. Otherwise it returns `None` and writes the report to the given file-like object (which must have a `.write()` method at a minimum).

indent : int

The number of 4 space tabs to indent the report.

Returns

report : str or None

HeaderDiff

```
class astropy.io.fits.HeaderDiff(a, b, ignore_keywords=[], ignore_comments=[], tolerance=0.0,
                                ignore_blanks=True, ignore_blank_cards=True)
```

Bases: `astropy.io.fits.diff._BaseDiff`

Diff two `Header` objects.

`HeaderDiff` objects have the following diff attributes:

- `diff_keyword_count`: If the two headers contain a different number of keywords, this contains a 2-tuple of the keyword count for each header.
- `diff_keywords`: If either header contains one or more keywords that don't appear at all in the other header, this contains a 2-tuple consisting of a list of the keywords only appearing in header a, and a list of the keywords only appearing in header b.
- `diff_duplicate_keywords`: If a keyword appears in both headers at least once, but contains a different number of duplicates (for example, a different number of HISTORY cards in each header), an item is added to this dict with the keyword as the key, and a 2-tuple of the different counts of that keyword as the value. For example:

```
{'HISTORY': (20, 19)}
```

means that header a contains 20 HISTORY cards, while header b contains only 19 HISTORY cards.

- `diff_keyword_values`: If any of the common keyword between the two headers have different values, they appear in this dict. It has a structure similar to `diff_duplicate_keywords`, with the keyword as the key, and a 2-tuple of the different values as the value. For example:

```
{'NAXIS': (2, 3)}
```

means that the NAXIS keyword has a value of 2 in header a, and a value of 3 in header b. This excludes any keywords matched by the `ignore_keywords` list.

- `diff_keyword_comments`: Like `diff_keyword_values`, but contains differences between keyword comments.

`HeaderDiff` objects also have a `common_keywords` attribute that lists all keywords that appear in both headers.

See `FITSDiff` for explanations of the initialization parameters.

classmethod fromdiff (*other, a, b*)

Returns a new Diff object of a specific subclass from an existing diff object, passing on the values for any arguments they share in common (such as `ignore_keywords`).

For example:

```
>>> hdul1, hdul2 = HDUList(), HDUList()
>>> headera, headerb = Header(), Header()
>>> fd = FITSDiff(hdul1, hdul2, ignore_keywords=['*'])
>>> hd = HeaderDiff.fromdiff(fd, headera, headerb)
>>> list(hd.ignore_keywords)
['*']
```

identical

`True` if all the `.diff_*` attributes on this diff instance are empty, implying that no differences were found.

Any subclass of `_BaseDiff` must have at least one `.diff_*` attribute, which contains a non-empty value if and only if some difference was found between the two objects being compared.

report (*fileobj=None, indent=0*)

Generates a text report on the differences (if any) between two objects, and either returns it as a string or writes it to a file-like object.

Parameters

fileobj : file-like object or `None`, optional

If `None`, this method returns the report as a string. Otherwise it returns `None` and writes the report to the given file-like object (which must have a `.write()` method at a minimum).

indent : int

The number of 4 space tabs to indent the report.

Returns

report : str or `None`

ImageDataDiff

class `astropy.io.fits.ImageDataDiff` (*a, b, numdiffs=10, tolerance=0.0*)

Bases: `astropy.io.fits.diff._BaseDiff`

Diff two image data arrays (really any array from a PRIMARY HDU or an IMAGE extension HDU, though the data unit is assumed to be “pixels”).

`ImageDataDiff` objects have the following diff attributes:

- `diff_dimensions`: If the two arrays contain either a different number of dimensions or different sizes in any dimension, this contains a 2-tuple of the shapes of each array. Currently no further comparison is performed on images that don't have the exact same dimensions.
- `diff_pixels`: If the two images contain any different pixels, this contains a list of 2-tuples of the array index where the difference was found, and another 2-tuple containing the different values. For example, if the pixel at (0, 0) contains different values this would look like:

```
[(0, 0), (1.1, 2.2)]
```

where 1.1 and 2.2 are the values of that pixel in each array. This array only contains up to `self.numdiffs` differences, for storage efficiency.

- `diff_total`: The total number of different pixels found between the arrays. Although `diff_pixels` does not necessarily contain all the different pixel values, this can be used to get a count of the total number of differences found.
- `diff_ratio`: Contains the ratio of `diff_total` to the total number of pixels in the arrays.

See `FITSDiff` for explanations of the initialization parameters.

classmethod `fromdiff` (*other, a, b*)

Returns a new `Diff` object of a specific subclass from an existing `diff` object, passing on the values for any arguments they share in common (such as `ignore_keywords`).

For example:

```
>>> hdul1, hdul2 = HDUList(), HDUList()
>>> headera, headerb = Header(), Header()
>>> fd = FITSDiff(hdul1, hdul2, ignore_keywords=['*'])
>>> hd = HeaderDiff.fromdiff(fd, headera, headerb)
>>> list(hd.ignore_keywords)
['*']
```

identical

`True` if all the `.diff_*` attributes on this `diff` instance are empty, implying that no differences were found.

Any subclass of `_BaseDiff` must have at least one `.diff_*` attribute, which contains a non-empty value if and only if some difference was found between the two objects being compared.

report (*fileobj=None, indent=0*)

Generates a text report on the differences (if any) between two objects, and either returns it as a string or writes it to a file-like object.

Parameters

fileobj : file-like object or `None`, optional

If `None`, this method returns the report as a string. Otherwise it returns `None` and writes the report to the given file-like object (which must have a `.write()` method at a minimum).

indent : int

The number of 4 space tabs to indent the report.

Returns

report : str or `None`

RawDataDiff

class `astropy.io.fits.RawDataDiff` (*a, b, numdiffs=10*)

Bases: `astropy.io.fits.ImageDataDiff`

`RawDataDiff` is just a special case of `ImageDataDiff` where the images are one-dimensional, and the data is treated as a 1-dimensional array of bytes instead of pixel values. This is used to compare the data of two non-standard extension HDUs that were not recognized as containing image or table data.

`ImageDataDiff` objects have the following `diff` attributes:

- `diff_dimensions`: Same as the `diff_dimensions` attribute of `ImageDataDiff` objects. Though the “dimension” of each array is just an integer representing the number of bytes in the data.
- `diff_bytes`: Like the `diff_pixels` attribute of `ImageDataDiff` objects, but renamed to reflect the minor semantic difference that these are raw bytes and not pixel values. Also the indices are integers instead of tuples.
- `diff_total` and `diff_ratio`: Same as `ImageDataDiff`.

See `FITSDiff` for explanations of the initialization parameters.

classmethod fromdiff (*other, a, b*)

Returns a new Diff object of a specific subclass from an existing diff object, passing on the values for any arguments they share in common (such as `ignore_keywords`).

For example:

```
>>> hdul1, hdul2 = HDUList(), HDUList()
>>> headera, headerb = Header(), Header()
>>> fd = FITSDiff(hdul1, hdul2, ignore_keywords=['*'])
>>> hd = HeaderDiff.fromdiff(fd, headera, headerb)
>>> list(hd.ignore_keywords)
['*']
```

identical

`True` if all the `.diff_*` attributes on this diff instance are empty, implying that no differences were found.

Any subclass of `_BaseDiff` must have at least one `.diff_*` attribute, which contains a non-empty value if and only if some difference was found between the two objects being compared.

report (*fileobj=None, indent=0*)

Generates a text report on the differences (if any) between two objects, and either returns it as a string or writes it to a file-like object.

Parameters

fileobj : file-like object or None, optional

If `None`, this method returns the report as a string. Otherwise it returns `None` and writes the report to the given file-like object (which must have a `.write()` method at a minimum).

indent : int

The number of 4 space tabs to indent the report.

Returns

report : str or None

TableDataDiff

class `astropy.io.fits.TableDataDiff` (*a, b, ignore_fields=[], numdiffs=10, tolerance=0.0*)

Bases: `astropy.io.fits.diff._BaseDiff`

Diff two table data arrays. It doesn't matter whether the data originally came from a binary or ASCII table—the data should be passed in as a recarray.

`TableDataDiff` objects have the following diff attributes:

- `diff_column_count`: If the tables being compared have different numbers of columns, this contains a 2-tuple of the column count in each table. Even if the tables have different column counts, an attempt is still made to compare any columns they have in common.
- `diff_columns`: If either table contains columns unique to that table, either in name or format, this contains a 2-tuple of lists. The first element is a list of columns (these are full `Column` objects) that appear only in table a. The second element is a list of tables that appear only in table b. This only lists columns with different column definitions, and has nothing to do with the data in those columns.
- `diff_column_names`: This is like `diff_columns`, but lists only the names of columns unique to either table, rather than the full `Column` objects.

- `diff_column_attributes`: Lists columns that are in both tables but have different secondard attributes, such as TUNIT or TDISP. The format is a list of 2-tuples: The first a tuple of the column name and the attribute, the second a tuple of the different values.
- `diff_values`: `TableDataDiff` compares the data in each table on a column-by-column basis. If any different data is found, it is added to this list. The format of this list is similar to the `diff_pixels` attribute on `ImageDataDiff` objects, though the “index” consists of a (column_name, row) tuple. For example:

```
[('TARGET', 0), ('NGC1001', 'NGC1002')]
```

shows that the tables contain different values in the 0-th row of the ‘TARGET’ column.

- `diff_total` and `diff_ratio`: Same as `ImageDataDiff`.

`TableDataDiff` objects also have a `common_columns` attribute that lists the `Column` objects for columns that are identical in both tables, and a `common_column_names` attribute which contains a set of the names of those columns.

See `FITSDiff` for explanations of the initialization parameters.

classmethod `fromdiff` (*other, a, b*)

Returns a new `Diff` object of a specific subclass from an existing `diff` object, passing on the values for any arguments they share in common (such as `ignore_keywords`).

For example:

```
>>> hdul1, hdul2 = HDUList(), HDUList()
>>> headera, headerb = Header(), Header()
>>> fd = FITSDiff(hdul1, hdul2, ignore_keywords=['*'])
>>> hd = HeaderDiff.fromdiff(fd, headera, headerb)
>>> list(hd.ignore_keywords)
['*']
```

identical

`True` if all the `.diff_*` attributes on this `diff` instance are empty, implying that no differences were found.

Any subclass of `_BaseDiff` must have at least one `.diff_*` attribute, which contains a non-empty value if and only if some difference was found between the two objects being compared.

report (*fileobj=None, indent=0*)

Generates a text report on the differences (if any) between two objects, and either returns it as a string or writes it to a file-like object.

Parameters

fileobj : file-like object or `None`, optional

If `None`, this method returns the report as a string. Otherwise it returns `None` and writes the report to the given file-like object (which must have a `.write()` method at a minimum).

indent : int

The number of 4 space tabs to indent the report.

Returns

report : str or `None`

14.5.9 Verification options

There are 5 options for the `output_verify` argument of the following methods of `HDUList`: `close()`, `writeto()`, and `flush()`, or the `:meth:~_BaseHDU.writeto` method on any HDU object. In these cases, the verification option is passed to a `:meth:verify` call within these methods.

exception

This option will raise an exception if any FITS standard is violated. This is the default option for output (i.e. when `writeto()`, `close()`, or `flush()` is called. If a user wants to overwrite this default on output, the other options listed below can be used.

ignore

This option will ignore any FITS standard violation. On output, it will write the HDU List content to the output FITS file, whether or not it is conforming to FITS standard.

The `ignore` option is useful in these situations, for example:

1. An input FITS file with non-standard is read and the user wants to copy or write out after some modification to an output file. The non-standard will be preserved in such output file.
2. A user wants to create a non-standard FITS file on purpose, possibly for testing purpose.

No warning message will be printed out. This is like a silent `warn` (see below) option.

fix

This option will try to fix any FITS standard violations. It is not always possible to fix such violations. In general, there are two kinds of FITS standard violation: fixable and not fixable. For example, if a keyword has a floating number with an exponential notation in lower case ‘e’ (e.g. `1.23e11`) instead of the upper case ‘E’ as required by the FITS standard, it’s a fixable violation. On the other hand, a keyword name like `P.I.` is not fixable, since it will not know what to use to replace the disallowed periods. If a violation is fixable, this option will print out a message noting it is fixed. If it is not fixable, it will throw an exception.

The principle behind the fixing is do no harm. For example, it is plausible to ‘fix’ a `Card` with a keyword name like `P.I.` by deleting it, but Astropy will not take such action to hurt the integrity of the data.

Not all fixes may be the “correct” fix, but at least Astropy will try to make the fix in such a way that it will not throw off other FITS readers.

silentfix

Same as `fix`, but will not print out informative messages. This may be useful in a large script where the user does not want excessive harmless messages. If the violation is not fixable, it will still throw an exception.

warn

This option is the same as the `ignore` option but will send warning messages. It will not try to fix any FITS standard violations whether fixable or not.

ASCII TABLES (`ASTROPY.IO.ASCII`)

15.1 Introduction

`astropy.io.ascii` provides methods for reading and writing a wide range of ASCII data table formats via built-in *Extension Reader classes*. The emphasis is on flexibility and ease of use.

The following shows a few of the ASCII formats that are available, while the section on [Supported formats](#) contains the full list.

- `Basic`: basic table with customizable delimiters and header configurations
- `Cds`: [CDS format table](#) (also Vizier and ApJ machine readable tables)
- `Daophot`: table from the IRAF DAOPHOT package
- `FixedWidth`: table with fixed-width columns (see also [Fixed-width Gallery](#))
- `Ipac`: [IPAC format table](#)
- `HTML`: HTML format table contained in a `<table>` tag
- `Latex`: LaTeX table with `datavalue` in the `tabular` environment
- `Rdb`: tab-separated values with an extra line after the column definition line
- `SExtractor`: [SExtractor format table](#)

The `astropy.io.ascii` package is built on a modular and extensible class structure with independent *Base class elements* so that new formats can be easily accommodated.

Note: It is also possible to use the functionality from `astropy.io.ascii` through a higher-level interface in the `astropy.table` package. See [Unified file read/write interface](#) for more details.

15.2 Getting Started

15.2.1 Reading Tables

The majority of commonly encountered ASCII tables can be easily read with the `read()` function. Assume you have a file named `sources.dat` with the following contents:

```
obsid redshift X Y object
3102 0.32 4167 4085 Q1250+568-A
877 0.22 4378 3892 "Source 82"
```

This table can be read with the following:

```
>>> from astropy.io import ascii
>>> data = ascii.read("sources.dat")
>>> print data
obsid redshift X Y object
-----
 3102      0.32 4167 4085 Q1250+568-A
   877      0.22 4378 3892 Source 82
```

The first argument to the `read()` function can be the name of a file, a string representation of a table, or a list of table lines. By default `read()` will try to guess the table format by trying all the supported formats. If this does not work (for unusually formatted tables) then one needs give `astropy.io.ascii` additional hints about the format, for example:

```
>>> lines = ['objID                & osrcid                & xsrcid                ',
...         '----- & ----- & -----',
...         '                277955213 & S000.7044P00.7513 & XS04861B6_005',
...         '                889974380 & S002.9051P14.7003 & XS03957B7_004']
>>> data = ascii.read(lines, data_start=2, delimiter='&')
>>> print(data)
  objID                osrcid                xsrcid
-----
277955213 S000.7044P00.7513 XS04861B6_005
889974380 S002.9051P14.7003 XS03957B7_004
```

If the format of a file is known (e.g. it is a fixed width table or an IPAC table), then it is more efficient and reliable to provide a value for the `format` argument from one of the values in the supported formats. For example:

```
>>> data = ascii.read(lines, format='fixed_width_two_line', delimiter='&')
```

15.2.2 Writing Tables

The `write()` function provides a way to write a data table as a formatted ASCII table. For example the following writes a table as a simple space-delimited file:

```
>>> import numpy as np
>>> from astropy.table import Table
>>> x = np.array([1, 2, 3])
>>> y = x ** 2
>>> data = Table([x, y], names=['x', 'y'])
>>> ascii.write(data, 'values.dat')
```

The `values.dat` file will then contain:

```
x y
1 1
2 4
3 9
```

All of the input Reader formats supported by `astropy.io.ascii` for reading are also supported for writing. This provides a great deal of flexibility in the format for writing. The example below writes the data as a LaTeX table, using the option to send the output to `sys.stdout` instead of a file:

```
>>> import sys
>>> ascii.write(data, sys.stdout, format='latex')
\begin{table}
\begin{tabular}{cc}
x & y \\
1 & 1 \\
```

```

2 & 4 \\
3 & 9 \\
\end{tabular}
\end{table}

```

15.3 Supported formats

A full list of the supported `format` values and corresponding format types for ASCII tables is given below. The `Write` column indicates which formats support write functionality.

Format	Write	Description
<code>aastex</code>	Yes	<code>AASTeX</code> : AAS _T E _X deluxetable used for AAS journals
<code>basic</code>	Yes	<code>Basic</code> : Basic table with custom delimiters
<code>cds</code>		<code>Cds</code> : CDS format table
<code>commented_header</code>	Yes	<code>CommentedHeader</code> : Column names in a commented line
<code>csv</code>	Yes	<code>Csv</code> : Basic table with comma-separated values
<code>daophot</code>		<code>Daophot</code> : IRAF DAOPHOT format table
<code>fixed_width</code>	Yes	<code>FixedWidth</code> : Fixed width
<code>fixed_width_no_header</code>	Yes	<code>FixedWidthNoHeader</code> : Fixed width with no header
<code>fixed_width_two_line</code>	Yes	<code>FixedWidthTwoLine</code> : Fixed width with second header line
<code>html</code>	Yes	<code>HTML</code> : HTML format table
<code>ipac</code>	Yes	<code>Ipac</code> : IPAC format table
<code>latex</code>	Yes	<code>Latex</code> : LaTeX table
<code>no_header</code>	Yes	<code>NoHeader</code> : Basic table with no headers
<code>rdb</code>	Yes	<code>Rdb</code> : Tab-separated with a type definition header line
<code>sextractor</code>		<code>SExtractor</code> : SExtractor format table
<code>tab</code>	Yes	<code>Tab</code> : Basic table with tab-separated values

15.4 Using `astropy.io.ascii`

The details of using `astropy.io.ascii` are provided in the following sections:

15.4.1 Reading tables

Reading tables

The majority of commonly encountered ASCII tables can be easily read with the `read()` function:

```

>>> from astropy.io import ascii
>>> data = ascii.read(table)

```

where `table` is the name of a file, a string representation of a table, or a list of table lines. By default `read()` will try to [guess the table format](#) by trying all the supported formats. If this does not work (for unusually formatted tables) then one needs give `astropy.io.ascii` additional hints about the format, for example:

```

>>> data = astropy.io.ascii.read('t/nls1_stackinfo.dbout', data_start=2, delimiter='|')
>>> data = astropy.io.ascii.read('t/simple.txt', quotechar='"')
>>> data = astropy.io.ascii.read('t/simple4.txt', format='no_header', delimiter='|')

```

The `read()` function accepts a number of parameters that specify the detailed table format. Different formats can define different defaults, so the descriptions below sometimes mention “typical” default values. This refers to the `Basic` format reader and other similar character-separated formats.

Parameters for `read()`

table

[input table] There are four ways to specify the table to be read:

- Name of a file (string)
- Single string containing all table lines separated by newlines
- File-like object with a callable `read()` method
- List of strings where each list element is a table line

The first two options are distinguished by the presence of a newline in the string. This assumes that valid file names will not normally contain a newline.

format

[file format (default='basic')] This specifies the top-level format of the ASCII table, for example if it is a basic character delimited table, fixed format table, or a CDS-compatible table, etc. The value of this parameter must be one of the *Supported formats*.

guess: try to guess table format (default=True)

If set to `True` then `read()` will try to guess the table format by cycling through a number of possible table format permutations and attempting to read the table in each case. See the [Guess table format](#) section for further details.

delimiter

[column delimiter string] A one-character string used to separate fields which typically defaults to the space character. Other common values might be “\s” (whitespace), “,” or “|” or “\t” (tab). A value of “\s” allows any combination of the tab and space characters to delimit columns.

comment

[regular expression defining a comment line in table] If the `comment` regular expression matches the beginning of a table line then that line will be discarded from header or data processing. For the `basic` format this defaults to “\s*#” (any whitespace followed by #).

quotechar

[one-character string to quote fields containing special characters] This specifies the quote character and will typically be either the single or double quote character. This is can be useful for reading text fields with spaces in a space-delimited table. The default is typically the double quote.

header_start

[line index for the header line not counting comment lines] This specifies in the line index where the header line will be found. Comment lines are not included in this count and the counting starts from 0 (first non-comment line has index=0). If set to `None` this indicates that there is no header line and the column names will be auto-generated. The default is dependent on the format.

data_start: line index for the start of data not counting comment lines

This specifies in the line index where the data lines begin where the counting starts from 0 and does not include comment lines. The default is dependent on the format.

data_end: line index for the end of data (can be negative to count from end)

If this is not `None` then it allows for excluding lines at the end that are not valid data lines. A negative value means to count from the end, so -1 would exclude the last line, -2 the last two lines, and so on.

converters: dict of data type converters

See the [Converters](#) section for more information.

names: list of names corresponding to each data column

Define the complete list of names for each data column. This will override names found in the header (if it exists). If not supplied then use names from the header or auto-generated names if there is no header.

include_names: list of names to include in output

From the list of column names found from the header or the `names` parameter, select for output only columns within this list. If not supplied then include all names.

exclude_names: list of names to exclude from output

Exclude these names from the list of output columns. This is applied *after* the `include_names` filtering. If not specified then no columns are excluded.

fill_values: list of fill value specifiers

Specify input table entries which should be masked in the output table because they are bad or missing. See the [Bad or missing values](#) section for more information and examples. The default is that any blank table values are treated as missing.

fill_include_names: list of column names, which are affected by fill_values.

If not supplied, then `fill_values` can affect all columns.

fill_exclude_names: list of column names, which are not affected by fill_values.

If not supplied, then `fill_values` can affect all columns.

Outputter: Outputter class

This converts the raw data tables value into the output object that gets returned by `read()`. The default is `TableOutputter`, which returns a `Table` object.

Inputter: Inputter class

This is generally not specified.

data_splitter: Splitter class to split data columns

header_splitter: Splitter class to split header columns

Reader

[Reader class (*deprecated* in favor of `format`)] This specifies the top-level format of the ASCII table, for example if it is a basic character delimited table, fixed format table, or a CDS-compatible table, etc. The value of this parameter must be a Reader class. For basic usage this means one of the built-in [Extension Reader classes](#).

Bad or missing values

ASCII data tables can contain bad or missing values. A common case is when a table contains blank entries with no available data, for example:

```
>>> weather_data = """
...   day,precip,type
...   Mon,1.5,rain
...   Tues,,
...   Wed,1.1,snow
...   """
```

By default `read()` will interpret blank entries as being bad/missing and output a masked Table with those entries masked out by setting the corresponding mask value set to `True`:

```
>>> dat = ascii.read(weather_data)
>>> print dat
day precip type
----
Mon      1.5 rain
Tues     --  --
Wed      1.1 snow
```

If you want to replace the masked (missing) values with particular values, set the masked column `fill_value` attribute and then get the “filled” version of the table. This looks like the following:

```
>>> dat['precip'].fill_value = -999
>>> dat['type'].fill_value = 'N/A'
>>> print dat.filled()
day precip type
----
Mon      1.5 rain
Tues -999.0 N/A
Wed      1.1 snow
```

ASCII tables may also have other indicators of bad or missing data. For example a table may contain string values that are not a valid representation of a number, e.g. “...”, or a table may have special values like `-999` that are chosen to indicate missing data. The `read()` function has a flexible system to accommodate these cases by marking specified character sequences in the input data as “missing data” during the conversion process. Whenever missing data is found then the output will be a masked table.

This is done with the `fill_values` keyword argument, which can be set to a single missing-value specification `<missing_spec>` or a list of `<missing_spec>` tuples:

```
fill_values = <missing_spec> | [<missing_spec1>, <missing_spec2>, ...]
<missing_spec> = (<match_string>, '0', <optional col name 1>, <optional col name 2>, ...)
```

When reading a table the second element of a `<missing_spec>` should always be the string `'0'`, otherwise you may get unexpected behavior¹. By default the `<missing_spec>` is applied to all columns unless column name strings are supplied. An alternate way to limit the columns is via the `fill_include_names` and `fill_exclude_names` keyword arguments in `read()`.

In the example below we read back the weather table after filling the missing values in with typical placeholders:

```
>>> table = ['day   precip   type',
...         ' Mon     1.5    rain',
...         'Tues  -999.0  N/A',
...         ' Wed     1.1    snow']
>>> t = ascii.read(table, fill_values=[('-999.0', '0', 'precip'), ('N/A', '0', 'type')])
>>> print t
day precip type
----
Mon      1.5 rain
Tues     --  --
Wed      1.1 snow
```

Note: The default in `read()` is `fill_values=(' ', '0')`. This marks blank entries as being missing for any data type (int, float, or string). If `fill_values` is explicitly set in the call to `read()` then the default behavior

¹ The requirement to put the `'0'` there is the legacy of an old interface which is maintained for backward compatibility and also to match the format of `fill_value` for reading with the format of `fill_value` used for writing tables. On reading, the second element of the `<missing_spec>` tuple can actually be an arbitrary string value which replaces occurrences of the `<match_string>` string in the input stream prior to type conversion. This ends up being the value “behind the mask”, which should never be directly accessed. Only the value `'0'` is neutral when attempting to detect the column data type and perform type conversion. For instance if you used `'nan'` for the `<match_string>` value then integer columns would wind up as float.

of marking blank entries as missing no longer applies. For instance setting `fill_values=None` will disable this auto-masking without setting any other fill values. This can be useful for a string column where one of values happens to be "".

Guess table format

If the `guess` parameter in `read()` is set to `True` (which is the default) then `read()` will try to guess the table format by cycling through a number of possible table format permutations and attempting to read the table in each case. The first format which succeeds and will be used to read the table. To succeed the table must be successfully parsed by the Reader and satisfy the following column requirements:

- At least two table columns
- No column names are a float or int number
- No column names begin or end with space, comma, tab, single quote, double quote, or a vertical bar (|).

These requirements reduce the chance for a false positive where a table is successfully parsed with the wrong format. A common situation is a table with numeric columns but no header row, and in this case `astropy.io.ascii` will auto-assign column names because of the restriction on column names that look like a number.

The order of guessing is shown by this Python code, where `Reader` is the class which actually implements reading the different file formats:

```
for Reader in (Rdb, Tab, Cds, Daophot, SExtractor, Ipac, Latex, AASTex, HTML):
    read(Reader=Reader)
for Reader in (CommentedHeader, Basic, NoHeader):
    for delimiter in ("|", ",", " ", "\\s"):
        for quotechar in ('"', "'"):
            read(Reader=Reader, delimiter=delimiter, quotechar=quotechar)
```

Note that the `FixedWidth` derived-readers are not included in the default guess sequence (this causes problems), so to read such tables one must explicitly specify the format with the `format` keyword.

If none of the guesses succeed in reading the table (subject to the column requirements) a final try is made using just the user-supplied parameters but without checking the column requirements. In this way a table with only one column or column names that look like a number can still be successfully read.

The guessing process respects any values of the `Reader`, `delimiter`, and `quotechar` parameters that were supplied to the `read()` function. Any guesses that would conflict are skipped. For example the call:

```
>>> data = ascii.read(table, Reader=ascii.NoHeader, quotechar="")
```

would only try the four delimiter possibilities, skipping all the conflicting `Reader` and `quotechar` combinations.

Guessing can be disabled in two ways:

```
import astropy.io.ascii
data = astropy.io.ascii.read(table)                # guessing enabled by default
data = astropy.io.ascii.read(table, guess=False)  # disable for this call
astropy.io.ascii.set_guess(False)                 # set default to False globally
data = astropy.io.ascii.read(table)                # guessing disabled
```

Converters

`astropy.io.ascii` converts the raw string values from the table into numeric data types by using converter functions such as the Python `int` and `float` functions. For example `int("5.0")` will fail while `float("5.0")` will succeed and return 5.0 as a Python float.

The default converters are:

```
default_converters = [astropy.io.ascii.convert_numpy(numpy.int),
                      astropy.io.ascii.convert_numpy(numpy.float),
                      astropy.io.ascii.convert_numpy(numpy.str)]
```

These take advantage of the `convert_numpy()` function which returns a 2-element tuple (`converter_func`, `converter_type`) as described in the previous section. The type provided to `convert_numpy()` must be a valid `numpy` type, for example `numpy.int`, `numpy.uint`, `numpy.int8`, `numpy.int64`, `numpy.float`, `numpy.float64`, `numpy.str`.

The default converters for each column can be overridden with the `converters` keyword:

```
>>> import numpy as np
>>> converters = {'col1': [ascii.convert_numpy(np.uint)],
...             'col2': [ascii.convert_numpy(np.float32)]}
>>> ascii.read('file.dat', converters=converters)
```

Advanced customization

Here we provide a few examples that demonstrate how to extend the base functionality to handle special cases. To go beyond these simple examples the best reference is to read the code for the existing *Extension Reader classes*.

Define a custom reader functionally

```
def read_rdb_table(table):
    reader = astropy.io.ascii.Basic()
    reader.header.splitter.delimiter = '\t'
    reader.data.splitter.delimiter = '\t'
    reader.header.splitter.process_line = None
    reader.data.splitter.process_line = None
    reader.data.start_line = 2

    return reader.read(table)
```

Define custom readers by class inheritance

```
# Note: Tab, Csv, and Rdb are included in astropy.io.ascii for convenience.
class Tab(astropy.io.ascii.Basic):
    def __init__(self):
        astropy.io.ascii.Basic.__init__(self)
        self.header.splitter.delimiter = '\t'
        self.data.splitter.delimiter = '\t'
        # Don't strip line whitespace since that includes tabs
        self.header.splitter.process_line = None
        self.data.splitter.process_line = None
        # Don't strip data value spaces since that is significant in TSV tables
        self.data.splitter.process_val = None
        self.data.splitter.skipinitialspace = False

class Rdb(astropy.io.ascii.Tab):
    def __init__(self):
        astropy.io.ascii.Tab.__init__(self)
        self.data.start_line = 2

class Csv(astropy.io.ascii.Basic):
    def __init__(self):
        astropy.io.ascii.Basic.__init__(self)
```

```

self.data.splitter.delimiter = ','
self.header.splitter.delimiter = ','
self.header.start_line = 0
self.data.start_line = 1

```

Create a custom splitter.process_val function

```

# The default process_val() normally just strips whitespace.
# In addition have it replace empty fields with -999.
def process_val(x):
    """Custom splitter process_val function: Remove whitespace at the beginning
    or end of value and substitute -999 for any blank entries."""
    x = x.strip()
    if x == '':
        x = '-999'
    return x

# Create an RDB reader and override the splitter.process_val function
rdb_reader = astropy.io.ascii.get_reader(Reader=astropy.io.ascii.Rdb)
rdb_reader.data.splitter.process_val = process_val

```

15.4.2 Writing tables

Writing tables

`astropy.io.ascii` is able to write ASCII tables out to a file or file-like object using the same class structure and basic user interface as for reading tables.

The `write()` function provides a way to write a data table as a formatted ASCII table. For example:

```

>>> import numpy as np
>>> from astropy.io import ascii
>>> x = np.array([1, 2, 3])
>>> y = x ** 2
>>> ascii.write([x, y], 'values.dat', names=['x', 'y'])

```

The `values.dat` file will then contain:

```

x y
1 1
2 4
3 9

```

Most of the input table *Supported formats* for reading are also available for writing. This provides a great deal of flexibility in the format for writing. The example below writes the data as a LaTeX table, using the option to send the output to `sys.stdout` instead of a file:

```

>>> ascii.write(data, format='latex')
\begin{table}
\begin{tabular}{cc}
x & y \\
1 & 1 \\
2 & 4 \\
3 & 9 \\
\end{tabular}
\end{table}

```

Input data format

The input `table` argument to `write()` can be any value that is supported for initializing a `Table` object. This is documented in detail in the *Constructing a table* section and includes creating a table with a list of columns, a dictionary of columns, or from `numpy` arrays (either structured or homogeneous). The sections below show a few examples.

Table or NumPy structured array An AstroPy `Table` object or a NumPy structured array (or record array) can serve as input to the `write()` function.

```
>>> from astropy.io import ascii
>>> from astropy.table import Table

>>> data = Table({'a': [1, 2, 3],
...              'b': [4.0, 5.0, 6.0]},
...              names=['a', 'b'])
>>> ascii.write(data)
a b
1 4.0
2 5.0
3 6.0

>>> data = np.array([(1, 2., 'Hello'), (2, 3., "World")],
...                 dtype=('i4,f4,a10'))
>>> ascii.write(data)
f0 f1 f2
1 2.0 Hello
2 3.0 World
```

The output of `astropy.io.ascii.read` is a `Table` or NumPy array data object that can be an input to the `write()` function.

```
>>> data = ascii.read('t/daophot.dat', format='daophot')
>>> ascii.write(data, 'space_delimited_table.dat')
```

List of lists A list of Python lists (or any iterable object) can be used as input:

```
>>> x = [1, 2, 3]
>>> y = [4, 5.2, 6.1]
>>> z = ['hello', 'world', '!!!']
>>> data = [x, y, z]

>>> ascii.write(data)
col0 col1 col2
1 4.0 hello
2 5.2 world
3 6.1 !!!
```

The data object does not contain information about the column names so `Table` has chosen them automatically. To specify the names, provide the `names` keyword argument. This example also shows excluding one of the columns from the output:

```
>>> ascii.write(data, names=['x', 'y', 'z'], exclude_names=['y'])
x z
1 hello
2 world
3 !!!
```

Dict of lists A dictionary containing iterable objects can serve as input to `write()`. Each dict key is taken as the column name while the value must be an iterable object containing the corresponding column values.

Since a Python dictionary is not ordered the output column order will be unpredictable unless the `names` argument is provided.

```
>>> data = {'x': [1, 2, 3],
...         'y': [4, 5.2, 6.1],
...         'z': ['hello', 'world', '!!!']}
>>> ascii.write(data, names=['x', 'y', 'z'])
x y z
1 4.0 hello
2 5.2 world
3 6.1 !!!
```

Parameters for `write()`

The `write()` function accepts a number of parameters that specify the detailed output table format. Each of the *Supported formats* is handled by a corresponding Writer class that can define different defaults, so the descriptions below sometimes mention “typical” default values. This refers to the `Basic` writer and other similar Writer classes.

Some output format Writer classes, e.g. `Latex` or `AASTeX` accept additional keywords, that can customize the output further. See the documentation of these classes for details.

output

[output specifier] There are two ways to specify the output for the write operation:

- Name of a file (string)
- File-like object (from `open()`, `StringIO`, etc)

table

[input table] Any value that is supported for initializing a `Table` object (see *Constructing a table*).

format

[output format (default='basic')] This specifies the format of the ASCII table to be written, for example if it is a basic character delimited table, fixed format table, or a CDS-compatible table, etc. The value of this parameter must be one of the *Supported formats*.

delimiter

[column delimiter string] A one-character string used to separate fields which typically defaults to the space character. Other common values might be “;”, “|” or “\t”.

comment

[string defining a comment line in table] For the `Basic` Writer this defaults to “#”. Which and how comments are written depends on the format chosen (e.g. `CommentedHeader` puts the comment symbol in the line with the column names).

formats: dict of data type converters

For each key (column name) use the given value to convert the column data to a string. If the format value is string-like then it is used as a Python format statement, e.g. “%0.2f” % value. If it is a callable function then that function is called with a single argument containing the column value to be converted. Example:

```
astropy.io.ascii.write(table, sys.stdout, formats={'XCENTER': '%12.1f',
                                                'YCENTER': lambda x: round(x, 1)},
```

names: list of names corresponding to each data column

Define the complete list of names for each data column. This will override names determined from the data table (if available). If not supplied then use names from the data table or auto-generated names.

include_names: list of names to include in output

From the list of column names found from the data table or the `names` parameter, select for output only columns within this list. If not supplied then include all names.

exclude_names: list of names to exclude from output

Exclude these names from the list of output columns. This is applied *after* the `include_names` filtering. If not specified then no columns are excluded.

fill_values: fill value specifier of lists

This can be used to fill missing values in the table or replace values with special meaning.

See the *Bad or missing values* section for more information on the syntax. The syntax is almost the same as when reading a table. There is a special value `astropy.io.ascii.masked` that is used to say “output this string for all masked values in a masked table (the default is to use a ‘--’):

```
>>> import sys
>>> from astropy.table import Table, Column, MaskedColumn
>>> from astropy.io import ascii
>>> t = Table([(1, 2), (3, 4)], names=('a', 'b'), masked=True)
>>> t['a'].mask = [True, False]
>>> ascii.write(t, sys.stdout)
a b
-- 3
2 4
>>> ascii.write(t, sys.stdout, fill_values=[(ascii.masked, 'N/A')])
a b
N/A 3
2 4
```

If no `fill_values` is applied for masked values in `astropy.io.ascii`, the default set with `numpy.ma.masked_print_option.set_display` applies (usually that is also ‘--’):

```
>>> ascii.write(t, sys.stdout, fill_values=[])
a b
-- 3
2 4
```

Note that when writing a table all values are converted to strings, before any value is replaced. Because `fill_values` only replaces cells that are an exact match to the specification, you need to provide the string representation (stripped of whitespace) for each value. For example, in the following commands `-99` is formatted with two digits after the comma, so we need to replace `-99.00` and not `-99`:

```
>>> t = Table([(-99, 2), (3, 4)], names=('a', 'b'))
>>> ascii.write(t, sys.stdout, fill_values = [('-99.00', 'no data')],
...           formats={'a': '%4.2f'})
a b
"no data" 3
2.00 4
```

Similarly, if you replace a value in a column that has a fixed length format, e.g. `'f4.2'`, then the string you want to replace must have the same number of characters, in the example above `fill_values=[('nan', ' N/A')]` would work.

fill_include_names: list of column names, which are affected by fill_values.

If not supplied, then `fill_values` can affect all columns.

fill_exclude_names: list of column names, which are not affected by fill_values.

If not supplied, then `fill_values` can affect all columns.

Writer

[Writer class (*deprecated* in favor of `format`)] This specifies the top-level format of the ASCII table to be

written, for example if it is a basic character delimited table, fixed format table, or a CDS-compatible table, etc. The value of this parameter must be a Writer class. For basic usage this means one of the built-in *Extension Reader classes*. Note: Reader classes and Writer classes are synonymous, in other words Reader classes can also write, but for historical reasons they are often called Reader classes.

15.4.3 Fixed-width Gallery

Fixed-width Gallery

Fixed-width tables are those where each column has the same width for every row in the table. This is commonly used to make tables easy to read for humans or FORTRAN codes. It also reduces issues with quoting and special characters, for example:

```
Col1   Col2   Col3 Col4
-----
 1.2   "hello"   1    a
 2.4   's worlds  2    2
```

There are a number of common variations in the formatting of fixed-width tables which `astropy.io.ascii` can read and write. The most significant difference is whether there is no header line (`FixedWidthNoHeader`), one header line (`FixedWidth`), or two header lines (`FixedWidthTwoLine`). Next, there are variations in the delimiter character, whether the delimiter appears on either end (“bookends”), and padding around the delimiter.

Details are available in the class API documentation, but the easiest way to understand all the options and their interactions is by example.

Reading

FixedWidth Nice, typical fixed format table

```
>>> from astropy.io import ascii
>>> table = """
... # comment (with blank line above)
... | Col1 | Col2 |
... | 1.2 | "hello" |
... | 2.4 | 's worlds|
... """
>>> ascii.read(table, format='fixed_width')
<Table rows=2 names=('Col1', 'Col2')>
array([(1.2, "hello"), (2..., "'s worlds")],
      dtype=[('Col1', '<f8'), ('Col2', 'S9')])
```

Typical fixed format table with col names provided

```
>>> table = """
... # comment (with blank line above)
... | Col1 | Col2 |
... | 1.2 | "hello" |
... | 2.4 | 's worlds|
... """
>>> ascii.read(table, format='fixed_width', names=('name1', 'name2'))
<Table rows=2 names=('name1', 'name2')>
array([(1.2, "hello"), (2..., "'s worlds")],
      dtype=[('name1', '<f8'), ('name2', 'S9')])
```

Weird input table with data values chopped by col extent

```
>>> table = """
...   Col1 | Col2 |
...   1.2      "hello"
...   2.4    sdf's worlds
...   """
>>> ascii.read(table, format='fixed_width')
<Table rows=2 names=('Col1', 'Col2')>
array([(1.2, 'hel'), (2..., "df's wo)],
      dtype=[('Col1', '<f8'), ('Col2', 'S7')])
```

Table with double delimiters

```
>>> table = """
... || Name || Phone || TCP||
... | John | 555-1234 |192.168.1.10X|
... | Mary | 555-2134 |192.168.1.12X|
... | Bob | 555-4527 | 192.168.1.9X|
...   """
>>> ascii.read(table, format='fixed_width')
<Table rows=3 names=('Name', 'Phone', 'TCP')>
array([('John', '555-1234', '192.168.1.10'),
      ('Mary', '555-2134', '192.168.1.12'),
      ('Bob', '555-4527', '192.168.1.9')],
      dtype=[('Name', 'S4'), ('Phone', 'S8'), ('TCP', 'S12')])
```

Table with space delimiter

```
>>> table = """
... Name --Phone- ----TCP-----
... John 555-1234 192.168.1.10
... Mary 555-2134 192.168.1.12
... Bob 555-4527 192.168.1.9
...   """
>>> ascii.read(table, format='fixed_width', delimiter=' ')
<Table rows=3 names=('Name', '--Phone-', '----TCP-----')>
array([('John', '555-1234', '192.168.1.10'),
      ('Mary', '555-2134', '192.168.1.12'),
      ('Bob', '555-4527', '192.168.1.9')],
      dtype=[('Name', 'S4'), ('--Phone-', 'S8'), ('----TCP-----', 'S12')])
```

Table with no header row and auto-column naming.

Use `header_start` and `data_start` keywords to indicate no header line.

```
>>> table = """
... | John | 555-1234 |192.168.1.10|
... | Mary | 555-2134 |192.168.1.12|
... | Bob | 555-4527 | 192.168.1.9|
...   """
>>> ascii.read(table, format='fixed_width',
...             header_start=None, data_start=0)
<Table rows=3 names=('col1', 'col2', 'col3')>
array([('John', '555-1234', '192.168.1.10'),
      ('Mary', '555-2134', '192.168.1.12'),
      ('Bob', '555-4527', '192.168.1.9')],
      dtype=[('col1', 'S4'), ('col2', 'S8'), ('col3', 'S12')])
```

Table with no header row and with col names provided.

Second and third rows also have hanging spaces after final “|”. Use `header_start` and `data_start` keywords to indicate

no header line.

```
>>> table = [" | John | 555-1234 |192.168.1.10|",
...          "| Mary | 555-2134 |192.168.1.12| ",
...          "| Bob | 555-4527 | 192.168.1.9| "]
>>> ascii.read(table, format='fixed_width',
...            header_start=None, data_start=0,
...            names=('Name', 'Phone', 'TCP'))
<Table rows=3 names=('Name', 'Phone', 'TCP')>
array([('John', '555-1234', '192.168.1.10'),
      ('Mary', '555-2134', '192.168.1.12'),
      ('Bob', '555-4527', '192.168.1.9')],
      dtype=[('Name', 'S4'), ('Phone', 'S8'), ('TCP', 'S12')])
```

FixedWidthNoHeader Table with no header row and auto-column naming. Use the `FixedWidthNoHeader` convenience class.

```
>>> table = """
... | John | 555-1234 |192.168.1.10|
... | Mary | 555-2134 |192.168.1.12|
... | Bob | 555-4527 | 192.168.1.9|
... """
>>> ascii.read(table, format='fixed_width_no_header')
<Table rows=3 names=('col1', 'col2', 'col3')>
array([('John', '555-1234', '192.168.1.10'),
      ('Mary', '555-2134', '192.168.1.12'),
      ('Bob', '555-4527', '192.168.1.9')],
      dtype=[('col1', 'S4'), ('col2', 'S8'), ('col3', 'S12')])
```

Table with no delimiter with column start and end values specified.

This uses the `col_starts` and `col_ends` keywords. Note that the `col_ends` values are inclusive so a position range of 0 to 5 will select the first 6 characters.

```
>>> table = """
... #   5   9   17 18   28   <== Column start / end indexes
... #   |   |   ||   |   <== Column separation positions
...   John 555- 1234 192.168.1.10
...   Mary 555- 2134 192.168.1.12
...   Bob  555- 4527 192.168.1.9
... """
>>> ascii.read(table, format='fixed_width_no_header',
...            names=('Name', 'Phone', 'TCP'),
...            col_starts=(0, 9, 18),
...            col_ends=(5, 17, 28),
...            )
<Table rows=3 names=('Name', 'Phone', 'TCP')>
array([('John', '555- 1234', '192.168.1.'),
      ('Mary', '555- 2134', '192.168.1.'),
      ('Bob', '555- 4527', '192.168.1')],
      dtype=[('Name', 'S4'), ('Phone', 'S9'), ('TCP', 'S10')])
```

FixedWidthTwoLine Typical fixed format table with two header lines with some cruft

```
>>> table = """
...   Col1   Col2
...   ----   -----
```

```

...     1.2xx"hello"
...     2.4   's worlds
...     ""
>>> ascii.read(table, format='fixed_width_two_line')
<Table rows=2 names=('Col1','Col2')>
array([(1.2, 'hello'), (2..., 's worlds)],
      dtype=[('Col1', '<f8'), ('Col2', 'S9')])

```

Restructured text table

```

>>> table = """
... =====
...   Col1   Col2
... =====
...   1.2   "hello"
...   2.4   's worlds
... =====
...   ""
>>> ascii.read(table, format='fixed_width_two_line',
...             header_start=1, position_line=2, data_end=-1)
<Table rows=2 names=('Col1','Col2')>
array([(1.2, 'hello'), (2..., 's worlds)],
      dtype=[('Col1', '<f8'), ('Col2', 'S9')])

```

Text table designed for humans and test having position line before the header line.

```

>>> table = """
... +-----+-----+
... | Col1 |   Col2   |
... +-----+-----+
... |  1.2 | "hello"  |
... |  2.4 | 's worlds|
... +-----+-----+
...   ""
>>> ascii.read(table, format='fixed_width_two_line', delimiter='+',
...             header_start=1, position_line=0, data_start=3, data_end=-1)
<Table rows=2 names=('Col1','Col2')>
array([(1.2, 'hello'), (2..., 's worlds)],
      dtype=[('Col1', '<f8'), ('Col2', 'S9')])

```

Writing

FixedWidth Define input values “dat“ for all write examples.

```

>>> table = """
... | Col1 |   Col2   | Col3 | Col4 |
... | 1.2 | "hello"  | 1    | a    |
... | 2.4 | 's worlds| 2    | 2    |
... ""
>>> dat = ascii.read(table, format='fixed_width')

```

Write a table as a normal fixed width table.

```

>>> ascii.write(dat, format='fixed_width')
| Col1 |   Col2   | Col3 | Col4 |
| 1.2 | "hello"  | 1    | a    |
| 2.4 | 's worlds| 2    | 2    |

```

Write a table as a fixed width table with no padding.

```
>>> ascii.write(dat, format='fixed_width', delimiter_pad=None)
|Col1|      Col2|Col3|Col4|
| 1.2|  "hello"|  1|  a|
| 2.4|'s worlds|  2|  2|
```

Write a table as a fixed width table with no bookend.

```
>>> ascii.write(dat, format='fixed_width', bookend=False)
Col1 |      Col2 | Col3 | Col4
 1.2 |  "hello" |  1 |  a
 2.4 | 's worlds |  2 |  2
```

Write a table as a fixed width table with no delimiter.

```
>>> ascii.write(dat, format='fixed_width', bookend=False, delimiter=None)
Col1      Col2  Col3  Col4
 1.2      "hello"    1    a
 2.4      's worlds    2    2
```

Write a table as a fixed width table with no delimiter and formatting.

```
>>> ascii.write(dat, format='fixed_width',
...             formats={'Col1': '%-8.3f', 'Col2': '%-15s'})
|      Col1 |      Col2 | Col3 | Col4 |
| 1.200    |  "hello"    |  1 |  a |
| 2.400    | 's worlds    |  2 |  2 |
```

FixedWidthNoHeader Write a table as a normal fixed width table.

```
>>> ascii.write(dat, format='fixed_width_no_header')
| 1.2 |  "hello" | 1 | a |
| 2.4 | 's worlds | 2 | 2 |
```

Write a table as a fixed width table with no padding.

```
>>> ascii.write(dat, format='fixed_width_no_header', delimiter_pad=None)
|1.2|  "hello"|1|a|
|2.4|'s worlds|2|2|
```

Write a table as a fixed width table with no bookend.

```
>>> ascii.write(dat, format='fixed_width_no_header', bookend=False)
1.2 |  "hello" | 1 | a
2.4 | 's worlds | 2 | 2
```

Write a table as a fixed width table with no delimiter.

```
>>> ascii.write(dat, format='fixed_width_no_header', bookend=False,
...             delimiter=None)
1.2      "hello"    1    a
2.4      's worlds    2    2
```

FixedWidthTwoLine Write a table as a normal fixed width table.

```
>>> ascii.write(dat, format='fixed_width_two_line')
Col1      Col2  Col3  Col4
```

```
-----  
1.2  "hello"    1    a  
2.4  's worlds  2    2
```

Write a table as a fixed width table with space padding and '=' position_char.

```
>>> ascii.write(dat, format='fixed_width_two_line',  
...             delimiter_pad=' ', position_char='=')  
Col1      Col2    Col3    Col4  
=====  =====  =====  =====  
1.2      "hello"    1        a  
2.4      's worlds  2        2
```

Write a table as a fixed width table with no bookend.

```
>>> ascii.write(dat, format='fixed_width_two_line', bookend=True, delimiter='|')  
|Col1|      Col2|Col3|Col4|  
|----|-----|----|----|  
| 1.2|  "hello"|  1|  a|  
| 2.4| 's worlds|  2|  2|
```

15.4.4 Base class elements

Base class elements

The key elements in `astropy.io.ascii` are:

- `Column`: Internal storage of column properties and data ()
- `Reader`: Base class to handle reading and writing tables.
- `Inputter`: Get the lines from the table input.
- `Splitter`: Split the lines into string column values.
- `Header`: Initialize output columns based on the table header or user input.
- `Data`: Populate column data from the table.
- `Outputter`: Convert column data to the specified output format, e.g. `numpy` structured array.

Each of these elements is an inheritable class with attributes that control the corresponding functionality. In this way the large number of tweakable parameters is modularized into manageable groups. Where it makes sense these attributes are actually functions that make it easy to handle special cases.

15.4.5 Extension Reader classes

Extension Reader classes

The following classes extend the base `BaseReader` functionality to handle reading and writing different table formats. Some, such as the `Basic` Reader class are fairly general and include a number of configurable attributes. Others such as `Cds` or `Daophot` are specialized to read certain well-defined but idiosyncratic formats.

- `AASTex`: AASTeX `deluxetable` used for AAS journals
- `Basic`: basic table with customizable delimiters and header configurations
- `Cds`: CDS format table (also Vizier and ApJ machine readable tables)
- `CommentedHeader`: column names given in a line that begins with the comment character

- `Daophot`: table from the IRAF DAOPHOT package
- `FixedWidth`: table with fixed-width columns (see also *Fixed-width Gallery*)
- `FixedWidthNoHeader`: table with fixed-width columns and no header
- `FixedWidthTwoLine`: table with fixed-width columns and a two-line header
- `HTML`: HTML format table contained in a `<table>` tag
- `Ipac`: IPAC format table
- `Latex`: LaTeX table with `datavalue` in the `tabular` environment
- `NoHeader`: basic table with no header where columns are auto-named
- `Rdb`: tab-separated values with an extra line after the column definition line
- `SExtractor`: SExtractor format table
- `Tab`: tab-separated values
- `Csv`: comma-separated values

15.5 Reference/API

15.5.1 `astropy.io.ascii` Module

An extensible ASCII table reader and writer.

Functions

<code>convert_numpy(numpy_type)</code>	Return a tuple (<code>converter_func</code> , <code>converter_type</code>).
<code>get_reader([Reader, Inputter, Outputter])</code>	Initialize a table reader allowing for common customizations.
<code>get_writer([Writer])</code>	Initialize a table writer allowing for common customizations.
<code>read(table[, guess])</code>	Read the input <code>table</code> and return the table.
<code>set_guess(guess)</code>	Set the default value of the <code>guess</code> parameter for <code>read()</code>
<code>write(table[, output, format, Writer])</code>	Write the input <code>table</code> to <code>filename</code> .

`convert_numpy`

`astropy.io.ascii.convert_numpy(numpy_type)`

Return a tuple (`converter_func`, `converter_type`). The converter function converts a list into a numpy array of the given `numpy_type`. This type must be a valid `numpy type`, e.g. `numpy.int`, `numpy.uint`, `numpy.int8`, `numpy.int64`, `numpy.float`, `numpy.float64`, `numpy.str`. The converter type is used to track the generic data type (`int`, `float`, `str`) that is produced by the converter function.

`get_reader`

`astropy.io.ascii.get_reader(Reader=None, Inputter=None, Outputter=None, **kwargs)`

Initialize a table reader allowing for common customizations. Most of the default behavior for various parameters is determined by the `Reader` class.

Parameters

- **Reader** – Reader class (DEPRECATED) (default= `Basic`)
- **Inputter** – Inputter class
- **Outputter** – Outputter class
- **delimiter** – column delimiter string
- **comment** – regular expression defining a comment line in table
- **quotechar** – one-character string to quote fields containing special characters
- **header_start** – line index for the header line not counting comment lines
- **data_start** – line index for the start of data not counting comment lines
- **data_end** – line index for the end of data (can be negative to count from end)
- **converters** – dict of converters
- **data_Splitter** – Splitter class to split data columns
- **header_Splitter** – Splitter class to split header columns
- **names** – list of names corresponding to each data column
- **include_names** – list of names to include in output (default=None selects all names)
- **exclude_names** – list of names to exclude from output (applied after `include_names`)
- **fill_values** – specification of fill values for bad or missing table values
- **fill_include_names** – list of names to include in `fill_values` (default=None selects all names)
- **fill_exclude_names** – list of names to exclude from `fill_values` (applied after `fill_include_names`)

`get_writer`

`astropy.io.ascii.get_writer` (*Writer=None, **kwargs*)

Initialize a table writer allowing for common customizations. Most of the default behavior for various parameters is determined by the `Writer` class.

Parameters

- **Writer** – Writer class (DEPRECATED) (default=‘‘`ascii.Basic`‘‘)
- **delimiter** – column delimiter string
- **write_comment** – string defining a comment line in table
- **quotechar** – one-character string to quote fields containing special characters
- **formats** – dict of format specifiers or formatting functions
- **strip_whitespace** – strip surrounding whitespace from column values (default=True)
- **names** – list of names corresponding to each data column
- **include_names** – list of names to include in output (default=None selects all names)
- **exclude_names** – list of names to exclude from output (applied after `include_names`)

read

`astropy.io.ascii.read` (*table*, *guess=None*, ***kwargs*)

Read the input `table` and return the table. Most of the default behavior for various parameters is determined by the Reader class.

Parameters

- **table** – input table (file name, file-like object, list of strings, or single newline-separated string)
- **guess** – try to guess the table format (default=True)
- **format** – input table format
- **Inputter** – Inputter class
- **Outputter** – Outputter class (default=TableOutputter)
- **delimiter** – column delimiter string
- **comment** – regular expression defining a comment line in table
- **quotechar** – one-character string to quote fields containing special characters
- **header_start** – line index for the header line not counting comment lines
- **data_start** – line index for the start of data not counting comment lines
- **data_end** – line index for the end of data (can be negative to count from end)
- **converters** – dict of converters
- **data_Splitter** – Splitter class to split data columns
- **header_Splitter** – Splitter class to split header columns
- **names** – list of names corresponding to each data column
- **include_names** – list of names to include in output (default=None selects all names)
- **exclude_names** – list of names to exclude from output (applied after `include_names`)
- **fill_values** – specification of fill values for bad or missing table values (default=(‘’, ‘0’))
- **fill_include_names** – list of names to include in `fill_values` (default=None selects all names)
- **fill_exclude_names** – list of names to exclude from `fill_values` (applied after `fill_include_names`)
- **Reader** – Reader class (DEPRECATED) (default=‘‘ascii.Basic‘‘)

set_guess

`astropy.io.ascii.set_guess` (*guess*)

Set the default value of the `guess` parameter for `read()`

Parameters

- guess** – New default `guess` value (True/False)

write

`astropy.io.ascii.write` (*table*, *output=None*, *format=None*, *Writer=None*, ***kwargs*)

Write the input *table* to *filename*. Most of the default behavior for various parameters is determined by the `Writer` class.

Parameters

- **table** – input table (Reader object, NumPy struct array, list of lists, etc)
- **output** – output [filename, file-like object] (default = `sys.stdout`)
- **format** – output format (default=‘basic‘)
- **delimiter** – column delimiter string
- **write_comment** – string defining a comment line in table
- **quotechar** – one-character string to quote fields containing special characters
- **formats** – dict of format specifiers or formatting functions
- **strip_whitespace** – strip surrounding whitespace from column values (default=True)
- **names** – list of names corresponding to each data column
- **include_names** – list of names to include in output (default=None selects all names)
- **exclude_names** – list of names to exclude from output (applied after `include_names`)
- **Writer** – Writer class (DEPRECATED) (default=‘ascii.Basic‘)

Classes

<code>AASTex(**kwargs)</code>	Write and read AASTeX tables.
<code>AllType</code>	Subclass of all other data types.
<code>BaseData()</code>	Base table data reader.
<code>BaseHeader()</code>	Base table header reader
<code>BaseInputter</code>	Get the lines from the table input and return a list of lines.
<code>BaseOutputter</code>	Output table as a dict of column objects keyed on column name.
<code>BaseReader()</code>	Class providing methods to read and write an ASCII table using the specified
<code>BaseSplitter</code>	Base splitter that uses python’s <code>split</code> method to do the work.
<code>Basic()</code>	Read a character-delimited table with a single header line at the top followed
<code>Cds([readme])</code>	Read a CDS format table.
<code>Column(name)</code>	Table column.
<code>CommentedHeader()</code>	Read a file where the column names are given in a line that begins with the he
<code>ContinuationLinesInputter</code>	Inputter where lines ending in <code>continuation_char</code> are joined with the su
<code>Csv()</code>	Read a CSV (comma-separated-values) file.
<code>Daophot()</code>	Read a DAOPhot file.
<code>DefaultSplitter()</code>	Default class to split strings into columns using python <code>csv</code> .
<code>FixedWidth([col_starts, col_ends, ...])</code>	Read or write a fixed width table with a single header line that defines column
<code>FixedWidthData()</code>	Base table data reader.
<code>FixedWidthHeader()</code>	Fixed width table header reader.
<code>FixedWidthNoHeader([col_starts, col_ends, ...])</code>	Read or write a fixed width table which has no header line.
<code>FixedWidthSplitter</code>	Split line based on fixed start and end positions for each <code>col</code> in <code>self.cols</code>
<code>FixedWidthTwoLine([position_line, ...])</code>	Read or write a fixed width table which has two header lines.
<code>FloatType</code>	Describes floating-point data.

Table 15.2 – continued from previous page

<code>HTML([htmldict])</code>	Read and write HTML tables.
<code>InconsistentTableError</code>	Indicates that an input table is inconsistent in some way.
<code>IntType</code>	Describes integer data.
<code>Ipac([definition, DBMS])</code>	Read or write an IPAC format table.
<code>Latex([ignore_latex_commands, latexdict, ...])</code>	Write and read LaTeX tables.
<code>NoHeader()</code>	Read a table with no header line.
<code>NoType</code>	Superclass for <code>StrType</code> and <code>NumType</code> classes.
<code>NumType</code>	Indicates that a column consists of numerical data.
<code>Rdb()</code>	Read a tab-separated file with an extra line after the column definition line.
<code>SExtractor()</code>	Read a SExtractor file.
<code>StrType</code>	Indicates that a column consists of text data.
<code>Tab()</code>	Read a tab-separated file.
<code>TableOutputter</code>	Output the table as an <code>astropy.table.Table</code> object.
<code>WhitespaceSplitter()</code>	

AASTex

class `astropy.io.ascii.AASTex` (***kwargs*)
 Bases: `astropy.io.ascii.Latex`

Write and read AASTeX tables.

This class implements some AASTeX specific commands. AASTeX is used for the AAS (American Astronomical Society) publications like ApJ, ApJL and AJ.

It derives from the `Latex` reader and accepts the same keywords. However, the keywords `header_start`, `header_end`, `data_start` and `data_end` in `latexdict` have no effect.

AllType

class `astropy.io.ascii.AllType`
 Bases: `astropy.io.ascii.StrType`, `astropy.io.ascii.FloatType`,
`astropy.io.ascii.IntType`

Subclass of all other data types.

This type is returned by `convert_numpy` if the given numpy type does not match `StrType`, `FloatType`, or `IntType`.

BaseData

class `astropy.io.ascii.BaseData`
 Bases: `object`

Base table data reader.

Parameters

- **start_line** – None, int, or a function of `lines` that returns None or int
- **end_line** – None, int, or a function of `lines` that returns None or int
- **comment** – Regular expression for comment lines
- **splitter_class** – Splitter class for splitting data lines into columns

Attributes Summary

```
comment
end_line
fill_exclude_names
fill_include_names
start_line
write_spacer_lines list() -> new empty list
```

Methods Summary

```
get_data_lines(lines) Set the data_lines attribute to the lines slice comprising the table data values.
get_str_vals() Return a generator that returns a list of column values (as strings) for each data line.
masks(cols) Set fill value for each column and then apply that fill value In the first step it is evaluated with value from
process_lines(lines) Strip out comment lines and blank lines from list of lines
write(lines)
```

Attributes Documentation

comment = None

end_line = None

fill_exclude_names = None

fill_include_names = None

start_line = None

write_spacer_lines = ['ASCII_TABLE_WRITE_SPACER_LINE']

Methods Documentation

get_data_lines (*lines*)

Set the `data_lines` attribute to the lines slice comprising the table data values.

get_str_vals ()

Return a generator that returns a list of column values (as strings) for each data line.

masks (*cols*)

Set fill value for each column and then apply that fill value

In the first step it is evaluated with value from `fill_values` applies to which column using `fill_include_names` and `fill_exclude_names`. In the second step all replacements are done for the appropriate columns.

process_lines (*lines*)

Strip out comment lines and blank lines from list of `lines`

Parameters**lines** – all lines in table**Returns**

list of lines

write (*lines*)**BaseHeader****class** `astropy.io.ascii.BaseHeader`Bases: `object`

Base table header reader

Parameters

- auto_format** – format string for auto-generating column names
- start_line** – None, int, or a function of `lines` that returns None or int
- comment** – regular expression for comment lines
- splitter_class** – Splitter class for splitting data lines into columns
- names** – list of names corresponding to each data column

Attributes Summary

<code>auto_format</code>	<code>str(object) -> string</code>
<code>colnames</code>	Return the column names of the table
<code>comment</code>	
<code>names</code>	
<code>start_line</code>	
<code>write_spacer_lines</code>	<code>list() -> new empty list</code>

Methods Summary

<code>get_col_type(col)</code>	
<code>get_cols(lines)</code>	Initialize the header Column objects from the table <code>lines</code> .
<code>get_type_map_key(col)</code>	
<code>process_lines(lines)</code>	Generator to yield non-comment lines
<code>update_meta(lines, meta)</code>	Extract any table-level metadata, e.g.
<code>write(lines)</code>	

Attributes Documentation**auto_format** = `'col%d'`**colnames**

Return the column names of the table

`comment = None`

`names = None`

`start_line = None`

`write_spacer_lines = ['ASCII_TABLE_WRITE_SPACER_LINE']`

Methods Documentation

`get_col_type (col)`

`get_cols (lines)`

Initialize the header Column objects from the table `lines`.

Based on the previously set Header attributes find or create the column names. Sets `self.cols` with the list of Columns.

Parameters

lines – list of table lines

Returns

None

`get_type_map_key (col)`

`process_lines (lines)`

Generator to yield non-comment lines

`update_meta (lines, meta)`

Extract any table-level metadata, e.g. keywords, comments, column metadata, from the table `lines` and update the OrderedDict `meta` in place. This base method does nothing.

`write (lines)`

BaseInputter

`class astropy.io.ascii.BaseInputter`

Bases: `object`

Get the lines from the table input and return a list of lines. The input table can be one of:

- File name
- String (newline separated) with all header and data lines (must have at least 2 lines)
- File-like object with `read()` method
- List of strings

Methods Summary

<code>get_lines(table)</code>	Get the lines from the <code>table</code> input.
<code>process_lines(lines)</code>	Process lines for subsequent use.

Methods Documentation

`get_lines` (*table*)

Get the lines from the `table` input.

Parameters

table – table input

Returns

list of lines

`process_lines` (*lines*)

Process lines for subsequent use. In the default case do nothing. This routine is not generally intended for removing comment lines or stripping whitespace. These are done (if needed) in the header and data line processing.

Override this method if something more has to be done to convert raw input lines to the table rows. For example the `ContinuationLinesInputter` derived class accounts for continuation characters if a row is split into lines.

BaseOutputter

class `astropy.io.ascii.BaseOutputter`

Bases: `object`

Output table as a dict of column objects keyed on column name. The table data are stored as plain python lists within the column objects.

Attributes Summary

`converters`

Attributes Documentation

`converters = {}`

BaseReader

class `astropy.io.ascii.BaseReader`

Bases: `object`

Class providing methods to read and write an ASCII table using the specified header, data, inputter, and outputter instances.

Typical usage is to instantiate a `Reader()` object and customize the `header`, `data`, `inputter`, and `outputter` attributes. Each of these is an object of the corresponding class.

There is one method `inconsistent_handler` that can be used to customize the behavior of `read()` in the event that a data row doesn't match the header. The default behavior is to raise an `InconsistentTableError`.

Attributes Summary

<code>comment_lines</code>	Return lines in the table that match header.comment regexp
<code>exclude_names</code>	
<code>include_names</code>	
<code>names</code>	
<code>strict_names</code>	bool(x) -> bool

Methods Summary

<code>inconsistent_handler(str_vals, ncols)</code>	Adjust or skip data entries if a row is inconsistent with the header.
<code>read(table)</code>	Read the <code>table</code> and return the results in a format determined by the <code>outputter</code> at
<code>write(table)</code>	Write <code>table</code> as list of strings.

Attributes Documentation

`comment_lines`

Return lines in the table that match header.comment regexp

`exclude_names = None`

`include_names = None`

`names = None`

`strict_names = False`

Methods Documentation

`inconsistent_handler` (*str_vals, ncols*)

Adjust or skip data entries if a row is inconsistent with the header.

The default implementation does no adjustment, and hence will always trigger an exception in `read()` any time the number of data entries does not match the header.

Note that this will *not* be called if the row already matches the header.

Parameters

- `str_vals` – A list of value strings from the current row of the table.
- `ncols` – The expected number of entries from the table header.

Returns

list of strings to be parsed into data entries in the output table. If the length of this list does

not match `ncols`, an exception will be raised in `read()`. Can also be `None`, in which case the row will be skipped.

read (*table*)

Read the `table` and return the results in a format determined by the `outputter` attribute.

The `table` parameter is any string or object that can be processed by the instance `inputter`. For the base `Inputter` class `table` can be one of:

- File name
- File-like object
- String (newline separated) with all header and data lines (must have at least 2 lines)
- List of strings

Parameters

table – table input

Returns

output table

write (*table*)

Write `table` as list of strings.

Parameters

table – input table data (`astropy.table.Table` object)

Returns

list of strings corresponding to ASCII table

BaseSplitter

class `astropy.io.ascii.BaseSplitter`

Bases: `object`

Base splitter that uses python's `split` method to do the work.

This does not handle quoted values. A key feature is the formulation of `__call__` as a generator that returns a list of the split line values at each iteration.

There are two methods that are intended to be overridden, first `process_line()` to do pre-processing on each input line before splitting and `process_val()` to do post-processing on each split string value. By default these apply the string `strip()` function. These can be set to another function via the instance attribute or be disabled entirely, for example:

```
reader.header.splitter.process_val = lambda x: x.lstrip()
reader.data.splitter.process_val = None
```

Parameters

delimiter – one-character string used to separate fields

Attributes Summary

`delimiter`

Methods Summary

<code>__call__(lines)</code>	
<code>join(vals)</code>	
<code>process_line(line)</code>	Remove whitespace at the beginning or end of line.
<code>process_val(val)</code>	Remove whitespace at the beginning or end of value.

Attributes Documentation

delimiter = None

Methods Documentation

`__call__(lines)`

`join(vals)`

process_line (*line*)

Remove whitespace at the beginning or end of line. This is especially useful for whitespace-delimited files to prevent spurious columns at the beginning or end.

process_val (*val*)

Remove whitespace at the beginning or end of value.

Basic

class `astropy.io.ascii.Basic`

Bases: `astropy.io.ascii.BaseReader`

Read a character-delimited table with a single header line at the top followed by data lines to the end of the table. Lines beginning with # as the first non-whitespace character are comments. This reader is highly configurable.

```
rdr = ascii.get_reader(Reader=ascii.Basic)
rdr.header.splitter.delimiter = ' '
rdr.data.splitter.delimiter = ' '
rdr.header.start_line = 0
rdr.data.start_line = 1
rdr.data.end_line = None
rdr.header.comment = r'\s*#'
rdr.data.comment = r'\s*#'
```

Example table:

```
# Column definition is the first uncommented line
# Default delimiter is the space character.
apples oranges pears
```

```
# Data starts after the header column definition, blank lines ignored
1 2 3
4 5 6
```

Cds

class `astropy.io.ascii.Cds` (*readme=None*)
 Bases: `astropy.io.ascii.BaseReader`

Read a CDS format table. See <http://vizier.u-strasbg.fr/doc/catstd.htx>. Example:

```
Table: Table name here
=====
Catalog reference paper
  Bibliography info here
=====
ADC_Keywords: Keyword ; Another keyword ; etc

Description:
  Catalog description here.
=====
Byte-by-byte Description of file: datafile3.txt
-----
  Bytes Format Units  Label  Explanations
-----
  1-   3 I3      ---    Index  Running identification number
  5-   6 I2      h      RAh    Hour of Right Ascension (J2000)
  8-   9 I2      min   RAm    Minute of Right Ascension (J2000)
 11-  15 F5.2    s      RAs    Second of Right Ascension (J2000)
-----
Note (1): A CDS file can contain sections with various metadata.
  Notes can be multiple lines.
Note (2): Another note.
-----
  1 03 28 39.09
  2 04 18 24.11
```

About parsing the CDS format

The CDS format consists of a table description and the table data. These can be in separate files as a `ReadMe` file plus data file(s), or combined in a single file. Different subsections within the description are separated by lines of dashes or equal signs (“-----” or “=====”). The table which specifies the column information must be preceded by a line starting with “Byte-by-byte Description of file:”.

In the case where the table description is combined with the data values, the data must be in the last section and must be preceded by a section delimiter line (dashes or equal signs only).

Basic usage

Use the `ascii.read()` function as normal, with an optional `readme` parameter indicating the CDS `ReadMe` file. If not supplied it is assumed that the header information is at the top of the given table. Examples:

```
>>> from astropy.io import ascii
>>> table = ascii.read("t/cds.dat")
>>> table = ascii.read("t/vizier/table1.dat", readme="t/vizier/ReadMe")
>>> table = ascii.read("t/cds/multi/lhs2065.dat", readme="t/cds/multi/ReadMe")
>>> table = ascii.read("t/cds/glob/lmxbrefs.dat", readme="t/cds/glob/ReadMe")
```

The table name and the CDS `ReadMe` file can be entered as URLs. This can be used to directly load tables from the Internet. For example, Vizier tables from the CDS:

```
>>> table = ascii.read("ftp://cdsarc.u-strasbg.fr/pub/cats/VII/253/snrs.dat",
...                   readme="ftp://cdsarc.u-strasbg.fr/pub/cats/VII/253/ReadMe")
```

If the header (ReadMe) and data are stored in a single file and there is content between the header and the data (for instance Notes), then the parsing process may fail. In this case you can instruct the reader to guess the actual start of the data by supplying `data_start='guess'` in the call to the `ascii.read()` function. You should verify that the output data table matches expectation based on the input CDS file.

Using a reader object

When Cds reader object is created with a `readme` parameter passed to it at initialization, then when the `read` method is executed with a table filename, the header information for the specified table is taken from the `readme` file. An `InconsistentTableError` is raised if the `readme` file does not have header information for the given table.

```
>>> readme = "t/vizier/ReadMe"
>>> r = ascii.get_reader(ascii.Cds, readme=readme)
>>> table = r.read("t/vizier/table1.dat")
>>> # table5.dat has the same ReadMe file
>>> table = r.read("t/vizier/table5.dat")
```

If no `readme` parameter is specified, then the header information is assumed to be at the top of the given table.

```
>>> r = ascii.get_reader(ascii.Cds)
>>> table = r.read("t/cds.dat")
>>> #The following gives InconsistentTableError, since no
>>> #readme file was given and table1.dat does not have a header.
>>> table = r.read("t/vizier/table1.dat")
Traceback (most recent call last):
...
InconsistentTableError: No CDS section delimiter found
Traceback (most recent call last):
...
InconsistentTableError: No CDS section delimiter found
```

Caveats:

- The Units and Explanations are available in the column `unit` and `description` attributes, respectively.
- The other metadata defined by this format is not available in the output table.

Methods Summary

<code>read(table)</code>
<code>write([table])</code> Not available for the Cds class (raises <code>NotImplementedError</code>)

Methods Documentation

read (*table*)

write (*table=None*)
Not available for the Cds class (raises `NotImplementedError`)

Column

```
class astropy.io.ascii.Column (name)
    Bases: object
```

Table column.

The key attributes of a Column object are:

- name** : column name
- type** : column type (NoType, StrType, NumType, FloatType, IntType)
- str_vals** : list of column values as strings
- data** : list of converted column values

CommentedHeader

class `astropy.io.ascii.CommentedHeader`

Bases: `astropy.io.ascii.BaseReader`

Read a file where the column names are given in a line that begins with the header comment character. `header_start` can be used to specify the line index of column names, and it can be a negative index (for example -1 for the last commented line). The default delimiter is the <space> character.:

```
# col1 col2 col3
# Comment line
1 2 3
4 5 6
```

ContinuationLinesInputter

class `astropy.io.ascii.ContinuationLinesInputter`

Bases: `astropy.io.ascii.BaseInputter`

Inputter where lines ending in `continuation_char` are joined with the subsequent line. Example:

```
col1 col2 col3
1      2 3
4 5      6
```

Attributes Summary

<code>continuation_char</code>	<code>str(object) -> string</code>
<code>no_continue</code>	
<code>replace_char</code>	<code>str(object) -> string</code>

Methods Summary

<code>process_lines(lines)</code>

Attributes Documentation

`continuation_char = '\'`

`no_continue = None`

```
replace_char = ''
```

Methods Documentation

process_lines (*lines*)

Csv

class `astropy.io.ascii.Csv`

Bases: `astropy.io.ascii.Basic`

Read a CSV (comma-separated-values) file.

Example:

```
num, ra, dec, radius, mag
1, 32.23222, 10.1211, 0.8, 18.1
2, 38.12321, -88.1321, 2.2, 17.0
```

Plain csv (comma separated value) files typically contain as many entries as there are columns on each line. In contrast, common spreadsheet editors stop writing if all remaining cells on a line are empty, which can lead to lines where the rightmost entries are missing. This Reader can deal with such files. Masked values (indicated by an empty '' field value when reading) are written out in the same way with an empty (') field. This is different from the typical default for `astropy.io.ascii` in which missing values are indicated by --.

Example:

```
num, ra, dec, radius, mag
1, 32.23222, 10.1211
2, 38.12321, -88.1321, 2.2, 17.0
```

Methods Summary

<code>inconsistent_handler(str_vals, ncols)</code>	Adjust row if it is too short.
--	--------------------------------

Methods Documentation

inconsistent_handler (*str_vals*, *ncols*)

Adjust row if it is too short.

If a data row is shorter than the header, add empty values to make it the right length. Note that this will *not* be called if the row already matches the header.

Parameters

- **str_vals** – A list of value strings from the current row of the table.
- **ncols** – The expected number of entries from the table header.

Returns

list of strings to be parsed into data entries in the output table.

Daophot

class `astropy.io.ascii.Daophot`

Bases: `astropy.io.ascii.BaseReader`

Read a DAOPHOT file. Example:

```
#K MERGERAD = INDEF scaleunit %-23.7g
#K IRAF = NOAO/IRAFV2.10EXPORT version %-23s
#K USER = davis name %-23s
#K HOST = tucana computer %-23s
#
#N ID XCENTER YCENTER MAG MERR MSKY NITER \
#U ## pixels pixels magnitudes magnitudes counts ## \
#F %-9d %-10.3f %-10.3f %-12.3f %-14.3f %-15.7g %-6d
#
#N SHARPNESS CHI PIER PERROR \
#U ## ## ## perrors \
#F %-23.3f %-12.3f %-6d %-13s
#
14 138.538 INDEF 15.461 0.003 34.85955 4 \
-0.032 0.802 0 No_error
```

The keywords defined in the #K records are available via the output table `meta` attribute:

```
>>> import os
>>> from astropy.io import ascii
>>> filename = os.path.join(ascii.__path__[0], 'tests/t/daophot.dat')
>>> data = ascii.read(filename)
>>> for name, keyword in data.meta['keywords'].items():
...     print(name, keyword['value'], keyword['units'], keyword['format'])
...
MERGERAD INDEF scaleunit %-23.7g
IRAF NOAO/IRAFV2.10EXPORT version %-23s
USER name %-23s
...
```

The unit and formats are available in the output table columns:

```
>>> for colname in data.colnames:
...     col = data[colname]
...     print(colname, col.unit, col.format)
...
ID None %-9d
XCENTER pixels %-10.3f
YCENTER pixels %-10.3f
...
```

Any column values of INDEF are interpreted as a missing value and will be masked out in the resultant table.

In case of multi-aperture daophot files containing repeated entries for the last row of fields, extra unique column names will be created by suffixing corresponding field names with numbers starting from 2 to N (where N is the total number of apertures). For example, first aperture radius will be RAPERT and corresponding magnitude will be MAG, second aperture radius will be RAPERT2 and corresponding magnitude will be MAG2, third aperture radius will be RAPERT3 and corresponding magnitude will be MAG3, and so on.

Methods Summary

`write([table])`

Methods Documentation

write (*table=None*)

DefaultSplitter

class `astropy.io.ascii.DefaultSplitter`

Bases: `astropy.io.ascii.BaseSplitter`

Default class to split strings into columns using python csv. The class attributes are taken from the csv Dialect class.

Typical usage:

```
# lines = ..
splitter = ascii.DefaultSplitter()
for col_vals in splitter(lines):
    for col_val in col_vals:
        ...
```

Parameters

- delimiter** – one-character string used to separate fields.
- doublequote** – control how instances of *quotechar* in a field are quoted
- escapechar** – character to remove special meaning from following character
- quotechar** – one-character string to quote fields containing special characters
- quoting** – control when quotes are recognised by the reader
- skipinitialspace** – ignore whitespace immediately following the delimiter

Attributes Summary

<code>delimiter</code>	<code>str(object) -> string</code>
<code>doublequote</code>	<code>bool(x) -> bool</code>
<code>escapechar</code>	
<code>quotechar</code>	<code>str(object) -> string</code>
<code>quoting</code>	<code>int(x[, base]) -> integer</code>
<code>skipinitialspace</code>	<code>bool(x) -> bool</code>

Methods Summary

<code>__call__(lines)</code>	Return an iterator over the table <i>lines</i> , where each iterator output is a list of the split line values.
------------------------------	---

Continued on next page

Table 15.19 – continued from previous page

```
join(vals)
process_line(line) Remove whitespace at the beginning or end of line.
```

Attributes Documentation**delimiter** = ‘ ‘**doublequote** = True**escapechar** = None**quotechar** = “”**quoting** = 0**skipinitialspace** = True**Methods Documentation****__call__** (*lines*)Return an iterator over the table *lines*, where each iterator output is a list of the split line values.**Parameters****lines** – list of table lines**Returns**

iterator

join (*vals*)**process_line** (*line*)

Remove whitespace at the beginning or end of line. This is especially useful for whitespace-delimited files to prevent spurious columns at the beginning or end. If splitting on whitespace then replace unquoted tabs with space first

FixedWidth

```
class astropy.io.ascii.FixedWidth (col_starts=None, col_ends=None, delimiter_pad=' ', book-
                                end=True)
```

Bases: `astropy.io.ascii.BaseReader`

Read or write a fixed width table with a single header line that defines column names and positions. Examples:

```
# Bar delimiter in header and data

| Col1 | Col2          | Col3 |
| 1.2  | hello there  | 3    |
| 2.4  | many words  | 7    |
```

```
# Bar delimiter in header only

Col1 |   Col2       | Col3
1.2   hello there   3
2.4   many words    7

# No delimiter with column positions specified as input

Col1      Col2Col3
1.2hello there   3
2.4many words    7
```

See the *Fixed-width Gallery* for specific usage examples.

Parameters

- **col_starts** – list of start positions for each column (0-based counting)
- **col_ends** – list of end positions (inclusive) for each column
- **delimiter_pad** – padding around delimiter when writing (default = None)
- **bookend** – put the delimiter at start and end of line when writing (default = False)

FixedWidthData

```
class astropy.io.ascii.FixedWidthData
```

Bases: `astropy.io.ascii.BaseData`

Base table data reader.

Parameters

- **start_line** – None, int, or a function of `lines` that returns None or int
- **end_line** – None, int, or a function of `lines` that returns None or int
- **comment** – Regular expression for comment lines
- **splitter_class** – Splitter class for splitting data lines into columns

Methods Summary

```
write(lines)
```

Methods Documentation

```
write (lines)
```

FixedWidthHeader

```
class astropy.io.ascii.FixedWidthHeader
```

Bases: `astropy.io.ascii.BaseHeader`

Fixed width table header reader.

The key settable class attributes are:

Parameters

- **auto_format** – format string for auto-generating column names
- **start_line** – None, int, or a function of `lines` that returns None or int
- **comment** – regular expression for comment lines
- **splitter_class** – Splitter class for splitting data lines into columns
- **position_line** – row index of line that specifies position (default = 1)
- **position_char** – character used to write the position line (default = “-”)
- **col_starts** – list of start positions for each column (0-based counting)
- **col_ends** – list of end positions (inclusive) for each column
- **delimiter_pad** – padding around delimiter when writing (default = None)
- **bookend** – put the delimiter at start and end of line when writing (default = False)

Attributes Summary

`position_line`

Methods Summary

<code>get_cols(lines)</code>	Initialize the header Column objects from the table <code>lines</code> .
<code>get_fixedwidth_params(line)</code>	Split <code>line</code> on the delimiter and determine column values and column start and end positions.
<code>get_line(lines, index)</code>	
<code>write(lines)</code>	

Attributes Documentation

position_line = None

Methods Documentation

get_cols (*lines*)

Initialize the header Column objects from the table `lines`.

Based on the previously set Header attributes find or create the column names. Sets `self.cols` with the list of Columns.

Parameters

lines – list of table lines

Returns

None

get_fixedwidth_params (*line*)

Split *line* on the delimiter and determine column values and column start and end positions. This might include null columns with zero length (e.g. for header row = "| col1 || col2 | col3 |" or header2_row = "-----"). The null columns are stripped out. Returns the values between delimiters and the corresponding start and end positions.

Parameters

line – input line

Returns

(vals, starts, ends)

get_line (*lines, index*)

write (*lines*)

FixedWidthNoHeader

```
class astropy.io.ascii.FixedWidthNoHeader (col_starts=None, col_ends=None, delimiter_pad='
', bookend=True)
```

Bases: `astropy.io.ascii.FixedWidth`

Read or write a fixed width table which has no header line. Column names are either input (`names` keyword) or auto-generated. Column positions are determined either by input (`col_starts` and `col_stops` keywords) or by splitting the first data line. In the latter case a `delimiter` is required to split the data line.

Examples:

```
# Bar delimiter in header and data
| 1.2 | hello there |    3 |
| 2.4 | many words  |    7 |

# Compact table having no delimiter and column positions specified as input
1.2hello there3
2.4many words 7
```

This class is just a convenience wrapper around the `FixedWidth` reader but with `header.start_line = None` and `data.start_line = 0`.

See the [Fixed-width Gallery](#) for specific usage examples.

Parameters

- **col_starts** – list of start positions for each column (0-based counting)
- **col_ends** – list of end positions (inclusive) for each column
- **delimiter_pad** – padding around delimiter when writing (default = None)
- **bookend** – put the delimiter at start and end of line when writing (default = False)

FixedWidthSplitter

```
class astropy.io.ascii.FixedWidthSplitter
```

Bases: `astropy.io.ascii.BaseSplitter`

Split line based on fixed start and end positions for each `col` in `self.cols`.

This class requires that the `Header` class will have defined `col.start` and `col.end` for each column. The reference to the `header.cols` gets put in the splitter object by the base `Reader.read()` function just in time for splitting data lines by a `data` object.

Note that the `start` and `end` positions are defined in the pythonic style so `line[start:end]` is the desired substring for a column. This splitter class does not have a hook for `process_lines` since that is generally not useful for fixed-width input.

Attributes Summary

<code>bookend</code>	<code>bool(x) -> bool</code>
<code>delimiter_pad</code>	<code>str(object) -> string</code>

Methods Summary

<code>__call__(lines)</code>
<code>join(vals, widths)</code>

Attributes Documentation

bookend = False

delimiter_pad = ''

Methods Documentation

__call__ (*lines*)

join (*vals, widths*)

FixedWidthTwoLine

```
class astropy.io.ascii.FixedWidthTwoLine (position_line=1, position_char='-',
                                          delimiter_pad=None, bookend=False)
```

Bases: `astropy.io.ascii.FixedWidth`

Read or write a fixed width table which has two header lines. The first header line defines the column names and the second implicitly defines the column positions. Examples:

```
# Typical case with column extent defined by ---- under column names.

coll      col2          <== header_start = 0
-----  -----
 1      bee flies      <== position_line = 1, position_char = "-"
 2      fish swims     <== data_start = 2
```

```
# Pretty-printed table

+-----+-----+
| Col1 | Col2 |
+-----+-----+
| 1.2 | "hello" |
| 2.4 | there world|
+-----+-----+
```

See the *Fixed-width Gallery* for specific usage examples.

Parameters

- **position_line** – row index of line that specifies position (default = 1)
- **position_char** – character used to write the position line (default = “-”)
- **delimiter_pad** – padding around delimiter when writing (default = None)
- **bookend** – put the delimiter at start and end of line when writing (default = False)

FloatType

class `astropy.io.ascii.FloatType`
Bases: `astropy.io.ascii.NumType`
Describes floating-point data.

HTML

class `astropy.io.ascii.HTML` (`htmldict={}`)
Bases: `astropy.io.ascii.BaseReader`

Read and write HTML tables.

In order to customize input and output, a dict of parameters may be passed to this class holding specific customizations.

htmldict : Dictionary of parameters for HTML input/output.

- **css**
[Customized styling] If present, this parameter will be included in a `<style>` tag and will define stylistic attributes of the output.
- **table_id**
[ID for the input table] If a string, this defines the HTML id of the table to be processed. If an integer, this specifies the index of the input table in the available tables. Unless this parameter is given, the reader will use the first table found in the input file.
- **multicol**
[Use multi-dimensional columns for output] The writer will output tuples as elements of multi-dimensional columns if this parameter is true, and if not then it will use the syntax `1.36583e-13 .. 1.36583e-13` for output. If not present, this parameter will be true by default.

Initialize classes for HTML reading and writing.

Methods Summary

<code>read(table)</code>	Read the <code>table</code> in HTML format and return a resulting <code>Table</code> .
<code>write(table)</code>	Return data in <code>table</code> converted to HTML as a list of strings.

Methods Documentation

`read(table)`

Read the `table` in HTML format and return a resulting `Table`.

`write(table)`

Return data in `table` converted to HTML as a list of strings.

InconsistentTableError

exception `astropy.io.ascii.InconsistentTableError`

Indicates that an input table is inconsistent in some way.

The default behavior of `BaseReader` is to throw an instance of this class if a data row doesn't match the header.

IntType

class `astropy.io.ascii.IntType`

Bases: `astropy.io.ascii.NumType`

Describes integer data.

Ipac

class `astropy.io.ascii.Ipac` (*definition='ignore', DBMS=False*)

Bases: `astropy.io.ascii.FixedWidth`

Read or write an IPAC format table. See http://irsa.ipac.caltech.edu/applications/DDGEN/Doc/ipac_tbl.html:

```

\name=value
\ Comment
| column1 | column2 | column3 | column4 | column5 |
| double | double  | int     | double  | char     |
| unit   | unit    | unit    | unit    | unit     |
| null   | null    | null    | null    | null     |
2.0978   29.09056  73765   2.06000  B8IVpMnHg

```

Or:

```

|----ra---|----dec---|---sao---|-----v---|----sptype-----|
  2.09708  29.09056   73765   2.06000   B8IVpMnHg

```

The comments and keywords defined in the header are available via the output table `meta` attribute:

```

>>> import os
>>> from astropy.io import ascii
>>> filename = os.path.join(ascii.__path__[0], 'tests/t/ipac.dat')

```

```
>>> data = ascii.read(filename)
>>> print(data.meta['comments'])
['This is an example of a valid comment']
>>> for name, keyword in data.meta['keywords'].items():
...     print(name, keyword['value'])
...
intval 1
floatval 2300.0
date Wed Sp 20 09:48:36 1995
key_continue IPAC keywords can continue across lines
```

Note that there are different conventions for characters occurring below the position of the | symbol in IPAC tables. By default, any character below a | will be ignored (since this is the current standard), but if you need to read files that assume characters below the | symbols belong to the column before or after the |, you can specify `definition='left'` or `definition='right'` respectively when reading the table (the default is `definition='ignore'`). The following examples demonstrate the different conventions:

•`definition='ignore'`:

```
| ra | dec |
| float | float |
1.2345 6.7890
```

•`definition='left'`:

```
| ra | dec |
| float | float |
1.2345 6.7890
```

•`definition='right'`:

```
| ra | dec |
| float | float |
1.2345 6.7890
```

Parameters

definition : str, optional

Specify the convention for characters in the data table that occur directly below the pipe (|) symbol in the header column definition:

- `'ignore'` - Any character beneath a pipe symbol is ignored (default)
- `'right'` - Character is associated with the column to the right
- `'left'` - Character is associated with the column to the left

DBMS : bool, optional

If true, this verifies that written tables adhere (semantically) to the [IPAC/DBMS](#) definition of IPAC tables. If `'False'` it only checks for the (less strict) [IPAC](#) definition.

Methods Summary

<code>write(table)</code>	Write <code>table</code> as list of strings.
---------------------------	--

Methods Documentation

write (*table*)

Write *table* as list of strings.

Parameters

table – input table data (astropy.table.Table object)

Returns

list of strings corresponding to ASCII table

Latex

```
class astropy.io.ascii.Latex (ignore_latex_commands=['hline', 'vspace', 'tableline'], latexdict={},
                             caption='', col_align=None)
```

Bases: `astropy.io.ascii.BaseReader`

Write and read LaTeX tables.

This class implements some LaTeX specific commands. Its main purpose is to write out a table in a form that LaTeX can compile. It is beyond the scope of this class to implement every possible LaTeX command, instead the focus is to generate a syntactically valid LaTeX tables.

This class can also read simple LaTeX tables (one line per table row, no `\multicolumn` or similar constructs), specifically, it can read the tables that it writes.

Reading a LaTeX table, the following keywords are accepted:

ignore_latex_commands :

Lines starting with these LaTeX commands will be treated as comments (i.e. ignored).

When writing a LaTeX table, the some keywords can customize the format. Care has to be taken here, because python interprets `\\` in a string as an escape character. In order to pass this to the output either format your strings as raw strings with the `r` specifier or use a double `\\\\\\`.

Examples:

```
caption = r'My table \label{mytable}'
caption = 'My table \\\label{mytable}'
```

latexdict : Dictionary of extra parameters for the LaTeX output

•**tabletype**

[used for first and last line of table.] The default is `\\begin{table}`. The following would generate a table, which spans the whole page in a two-column document:

```
ascii.write(data, sys.stdout, Writer = ascii.Latex,
            latexdict = {'tabletype': 'table*'})
```

•**tablealign**

[positioning of table in text.] The default is not to specify a position preference in the text. If, e.g. the alignment is `ht`, then the LaTeX will be `\\begin{table}[ht]`.

•**col_align**

[Alignment of columns] If not present all columns will be centered.

•**caption**

[Table caption (string or list of strings)] This will appear above the table as it is the standard in many scientific publications. If you prefer a caption below the table, just write the full LaTeX command as `latexdict['tablefoot'] = r'\caption{My table}'`

•preamble, header_start, header_end, data_start, data_end, tablefoot: Pure LaTeX

Each one can be a string or a list of strings. These strings will be inserted into the table without any further processing. See the examples below.

•units

[dictionary of strings] Keys in this dictionary should be names of columns. If present, a line in the LaTeX table directly below the column names is added, which contains the values of the dictionary. Example:

```
from astropy.io import ascii
data = {'name': ['bike', 'car'], 'mass': [75,1200], 'speed': [10, 130]}
ascii.write(data, Writer=ascii.Latex,
            latexdict = {'units': {'mass': 'kg', 'speed': 'km/h'}})
```

If the column has no entry in the units dictionary, it defaults to ' '.

Run the following code to see where each element of the dictionary is inserted in the LaTeX table:

```
from astropy.io import ascii
data = {'cola': [1,2], 'colb': [3,4]}
ascii.write(data, Writer=ascii.Latex, latexdict=ascii.latex.latexdicts['template'])
```

Some table styles are predefined in the dictionary `ascii.latex.latexdicts`. The following generates in table in style preferred by A&A and some other journals:

```
ascii.write(data, Writer=ascii.Latex, latexdict=ascii.latex.latexdicts['AA'])
```

As an example, this generates a table, which spans all columns and is centered on the page:

```
ascii.write(data, Writer=ascii.Latex, col_align='|lr|',
            latexdict={'preamble': r'\begin{center}',
                       'tablefoot': r'\end{center}',
                       'tabletype': 'table*'})
```

caption

[Set table caption] Shorthand for:

```
latexdict['caption'] = caption
```

col_align

[Set the column alignment.] If not present this will be auto-generated for centered columns. Shorthand for:

```
latexdict['col_align'] = col_align
```

Methods Summary

`write([table])`

Methods Documentation

write (*table=None*)

NoHeader

```
class astropy.io.ascii.NoHeader
    Bases: astropy.io.ascii.Basic
```

Read a table with no header line. Columns are autonamed using `header.auto_format` which defaults to “col%d”. Otherwise this reader the same as the `Basic` class from which it is derived. Example:

```
# Table data
1 2 "hello there"
3 4 world
```

NoType

```
class astropy.io.ascii.NoType
    Bases: object
```

Superclass for `StrType` and `NumType` classes.

This class is the default type of `Column` and provides a base class for other data types.

NumType

```
class astropy.io.ascii.NumType
    Bases: astropy.io.ascii.NoType
```

Indicates that a column consists of numerical data.

Rdb

```
class astropy.io.ascii.Rdb
    Bases: astropy.io.ascii.Tab
```

Read a tab-separated file with an extra line after the column definition line. The RDB format meets this definition. Example:

```
col1 <tab> col2 <tab> col3
N <tab> S <tab> N
1 <tab> 2 <tab> 5
```

In this reader the second line is just ignored.

SExtractor

```
class astropy.io.ascii.SExtractor
    Bases: astropy.io.ascii.BaseReader
```

Read a SExtractor file.

SExtractor is a package for faint-galaxy photometry. Bertin & Arnouts 1996, A&A Supp. 317, 393. <http://www.astromatic.net/software/sextractor>

Example:

```
# 1 NUMBER
# 2 ALPHA_J2000
# 3 DELTA_J2000
# 4 FLUX_RADIUS
# 7 MAG_AUTO [mag]
# 8 X2_IMAGE Variance along x [pixel**2]
# 9 X_MAMA Barycenter position along MAMA x axis [m**(-6)]
# 10 MU_MAX Peak surface brightness above background [mag * arcsec**(-2)]
1 32.23222 10.1211 0.8 1.2 1.4 18.1 1000.0 0.00304 -3.498
2 38.12321 -88.1321 2.2 2.4 3.1 17.0 1500.0 0.00908 1.401
```

Note the skipped numbers since `flux_radius` has 3 columns. The three `FLUX_RADIUS` columns will be named `FLUX_RADIUS`, `FLUX_RADIUS_1`, `FLUX_RADIUS_2`. Also note that a post-ID description (e.g. “Variance along x”) is optional and that units may be specified at the end of a line in brackets.

Methods Summary

```
read(table)
write([table])
```

Methods Documentation

read (*table*)

write (*table=None*)

StrType

class `astropy.io.ascii.StrType`
Bases: `astropy.io.ascii.NoType`
Indicates that a column consists of text data.

Tab

class `astropy.io.ascii.Tab`
Bases: `astropy.io.ascii.Basic`

Read a tab-separated file. Unlike the `Basic` reader, whitespace is not stripped from the beginning and end of either lines or individual column values.

Example:

```
col1 <tab> col2 <tab> col3
# Comment line
1 <tab> 2 <tab> 5
```

TableOutputter

class `astropy.io.ascii.TableOutputter`

Bases: `astropy.io.ascii.BaseOutputter`

Output the table as an `astropy.table.Table` object.

Attributes Summary

<code>default_converters</code>	list() -> new empty list
---------------------------------	--------------------------

Methods Summary

<code>__call__</code>	(cols, meta)
-----------------------	--------------

Attributes Documentation

`default_converters` = [(<function converter at 0x61af488>, <class 'astropy.io.ascii.core.IntType'>), (<function conve

Methods Documentation

`__call__` (*cols*, *meta*)

WhitespaceSplitter

class `astropy.io.ascii.WhitespaceSplitter`

Bases: `astropy.io.ascii.DefaultSplitter`

Methods Summary

<code>process_line</code>	(line) Replace tab with space within <i>line</i> while respecting quoted substrings
---------------------------	---

Methods Documentation

`process_line` (*line*)

Replace tab with space within *line* while respecting quoted substrings

Class Inheritance Diagram

VOTABLE XML HANDLING (ASTROPY . IO . VOTABLE)

16.1 Introduction

The `astropy.io.votable` subpackage converts VOTable XML files to and from Numpy record arrays.

16.2 Getting Started

16.2.1 Reading a VOTable file

To read in a VOTable file, pass a file path to `parse`:

```
from astropy.io.votable import parse
votable = parse("votable.xml")
```

`votable` is a `VOTableFile` object, which can be used to retrieve and manipulate the data and save it back out to disk.

VOTable files are made up of nested `RESOURCE` elements, each of which may contain one or more `TABLE` elements. The `TABLE` elements contain the arrays of data.

To get at the `TABLE` elements, one can write a loop over the resources in the VOTABLE file:

```
for resource in votable.resources:
    for table in resource.tables:
        # ... do something with the table ...
        pass
```

However, if the nested structure of the resources is not important, one can use `iter_tables` to return a flat list of all tables:

```
for table in votable.iter_tables():
    # ... do something with the table ...
    pass
```

Finally, if there is expected to be only one table in the file, it might be simplest to just use `get_first_table`:

```
table = votable.get_first_table()
```

Even easier, there is a convenience method to parse a VOTable file and return the first table all in one step:

```
from astropy.io.votable import parse_single_table
table = parse_single_table("votable.xml")
```

From a `Table` object, one can get the data itself in the `array` member variable:

```
data = table.array
```

This data is a Numpy record array.

The columns get their names from both the `ID` and `name` attributes of the `FIELD` elements in the `VOTABLE` file. For example, suppose we had a `FIELD` specified as follows:

```
<FIELD ID="Dec" name="dec_targ" datatype="char" ucd="POS_EQ_DEC_MAIN"
      unit="deg">
  <DESCRIPTION>
    representing the ICRS declination of the center of the image.
  </DESCRIPTION>
</FIELD>
```

Note: The mapping from `VOTable` `name` and `ID` attributes to Numpy `dtype` names and `titles` is highly confusing. In `VOTable`, `ID` is guaranteed to be unique, but is not required. `name` is not guaranteed to be unique, but is required.

In Numpy record dtypes, `names` are required to be unique and are required. `titles` are not required, and are not required to be unique.

Therefore, `VOTable`'s `ID` most closely maps to Numpy's `names`, and `VOTable`'s `name` most closely maps to Numpy's `titles`. However, in some cases where a `VOTable` `ID` is not provided, a Numpy `name` will be generated based on the `VOTable` `name`. Unfortunately, `VOTable` fields do not have an attribute that is both unique and required, which would be the most convenient mechanism to uniquely identify a column.

This column of data can be extracted from the record array using:

```
>>> table.array['dec_targ']
array([17.15153360566, 17.15153360566, 17.15153360566, 17.1516686826,
       17.1516686826, 17.1516686826, 17.1536197136, 17.1536197136,
       17.1536197136, 17.15375479055, 17.15375479055, 17.15375479055,
       17.1553884541, 17.15539736932, 17.15539752176,
       17.25736014763,
       # ...
       17.2765703], dtype=object)
```

or equivalently:

```
>>> table.array['Dec']
array([17.15153360566, 17.15153360566, 17.15153360566, 17.1516686826,
       17.1516686826, 17.1516686826, 17.1536197136, 17.1536197136,
       17.1536197136, 17.15375479055, 17.15375479055, 17.15375479055,
       17.1553884541, 17.15539736932, 17.15539752176,
       17.25736014763,
       # ...
       17.2765703], dtype=object)
```

16.2.2 Building a new table from scratch

It is also possible to build a new table, define some field datatypes and populate it with data:

```
from astropy.io.votable.tree import VOTableFile, Resource, Table, Field

# Create a new VOTable file...
votable = VOTableFile()
```

```

# ...with one resource...
resource = Resource()
votable.resources.append(resource)

# ... with one table
table = Table(votable)
resource.tables.append(table)

# Define some fields
table.fields.extend([
    Field(votable, name="filename", datatype="char", arraysize="*"),
    Field(votable, name="matrix", datatype="double", arraysize="2x2")])

# Now, use those field definitions to create the numpy record arrays, with
# the given number of rows
table.create_arrays(2)

# Now table.array can be filled with data
table.array[0] = ('test1.xml', [[1, 0], [0, 1]])
table.array[1] = ('test2.xml', [[0.5, 0.3], [0.2, 0.1]])

# Now write the whole thing to a file.
# Note, we have to use the top-level votable file object
votable.to_xml("new_votable.xml")

```

16.2.3 Outputting a VOTable file

To save a VOTable file, simply call the `to_xml` method. It accepts either a string or Unicode path, or a Python file-like object:

```
votable.to_xml('output.xml')
```

There are a number of data storage formats supported by `astropy.io.votable`. The TABLEDATA format is XML-based and stores values as strings representing numbers. The BINARY format is more compact, and stores numbers in base64-encoded binary. VOTable version 1.3 adds the BINARY2 format, which allows for masking of any data type, including integers and bit fields which can not be masked in the older BINARY format. The storage format can be set on a per-table basis using the `format` attribute, or globally using the `set_all_tables_format` method:

```

votable.get_first_table().format = 'binary'
votable.set_all_tables_format('binary')
votable.to_xml('binary.xml')

```

16.3 Using `astropy.io.votable`

16.3.1 Standard compliance

`astropy.io.votable.tree.Table` supports the VOTable Format Definition Version 1.1, Version 1.2, and the Version 1.3 proposed recommendation. Some flexibility is provided to support the 1.0 draft version and other non-standard usage in the wild. To support these cases, set the keyword argument `pedantic` to `False` when parsing.

Note: Each warning and VOTABLE-specific exception emitted has a number and is documented in more detail in [Warnings](#) and [Exceptions](#).

Output always conforms to the 1.1, 1.2 or 1.3 spec, depending on the input.

Pedantic mode

Many VOTABLE files in the wild do not conform to the VOTABLE specification. If reading one of these files causes exceptions, you may turn off pedantic mode in `astropy.io.votable` by passing `pedantic=False` to the `parse` or `parse_single_table` functions:

```
from astropy.io.votable import parse
votable = parse("votable.xml", pedantic=False)
```

Note, however, that it is good practice to report these errors to the author of the application that generated the VOTABLE file to bring the file into compliance with the specification.

Even with `pedantic` turned off, many warnings may still be omitted. These warnings are all of the type `VOTableSpecWarning` and can be turned off using the standard Python `warnings` module.

16.3.2 Missing values

Any value in the table may be “missing”. `astropy.io.votable` stores a Numpy masked array in each `Table` instance. This behaves like an ordinary Numpy masked array, except for variable-length fields. For those fields, the datatype of the column is “object” and another Numpy masked array is stored there. Therefore, operations on variable length columns will not work – this is simply because variable length columns are not directly supported by Numpy masked arrays.

16.3.3 Datatype mappings

The datatype specified by a `FIELD` element is mapped to a Numpy type according to the following table:

VOTABLE type	Numpy type
boolean	b1
bit	b1
unsignedByte	u1
char (<i>variable length</i>)	O - In Python 2.x, a <code>str</code> object; in 3.x, a <code>bytes()</code> object.
char (<i>fixed length</i>)	S
unicodeChar (<i>variable length</i>)	O - In Python 2.x, a <code>unicode</code> object, in utf-16; in 3.x a <code>str</code> object
unicodeChar (<i>fixed length</i>)	U
short	i2
int	i4
long	i8
float	f4
double	f8
floatComplex	c8
doubleComplex	c16

If the field is a fixed size array, the data is stored as a Numpy fixed-size array.

If the field is a variable size array (that is `arraysize` contains a `*`), the cell will contain a Python list of Numpy values. Each value may be either an array or scalar depending on the `arraysize` specifier.

16.3.4 Examining field types

To look up more information about a field in a table, one can use the `get_field_by_id` method, which returns the `Field` object with the given ID. For example:

```
>>> field = table.get_field_by_id('Dec')
>>> field.datatype
'char'
>>> field.unit
'deg'
```

Note: Field descriptors should not be mutated. To change the set of columns, convert the `Table` to an `astropy.table.Table`, make the changes, and then convert it back.

16.3.5 Data serialization formats

VOTable supports a number of different serialization formats.

- **TABLEDATA** stores the data in pure XML, where the numerical values are written as human-readable strings.
- **BINARY** is a binary representation of the data, stored in the XML as an opaque `base64`-encoded blob.
- **BINARY2** was added in VOTable 1.3, and is identical to “BINARY”, except that it explicitly records the position of missing values rather than identifying them by a special value.
- **FITS** stores the data in an external FITS file. This serialization is not supported by the `astropy.io.votable` writer, since it requires writing multiple files.

The serialization format can be selected in two ways:

- 1) By setting the `format` attribute of a `astropy.io.votable.tree.Table` object:

```
votable.get_first_table().format = "binary"
votable.to_xml("new_votable.xml")
```

- 2) By overriding the format of all tables using the `tabledata_format` keyword argument when writing out a VOTable file:

```
votable.to_xml("new_votable.xml", tabledata_format="binary")
```

16.3.6 Converting to/from an `astropy.table.Table`

The VOTable standard does not map conceptually to an `astropy.table.Table`. However, a single table within the VOTable file may be converted to and from an `astropy.table.Table`:

```
from astropy.io.votable import parse_single_table
table = parse_single_table("votable.xml").to_table()
```

As a convenience, there is also a function to create an entire VOTable file with just a single table:

```
from astropy.io.votable import from_table, writeto
votable = from_table(table)
writeto(votable, "output.xml")
```

Note: By default, `to_table` will use the ID attribute from the files to create the column names for the `Table` object. However, it may be that you want to use the `name` attributes instead. For this, set the `use_names_over_ids` keyword to `True`. Note that since field names are not guaranteed to be unique in the VOTable specification, but

column names are required to be unique in Numpy structured arrays (and thus `astropy.table.Table` objects), the names may be renamed by appending numbers to the end in some cases.

16.3.7 Performance considerations

File reads will be moderately faster if the `TABLE` element includes an `nrows` attribute. If the number of rows is not specified, the record array must be resized repeatedly during load.

16.4 See Also

- [VOTable Format Definition Version 1.1](#)
- [VOTable Format Definition Version 1.2](#)
- [VOTable Format Definition Version 1.3, Proposed Recommendation](#)

16.5 Reference/API

16.5.1 `astropy.io.votable` Module

This package reads and writes data formats used by the Virtual Observatory (VO) initiative, particularly the VOTable XML format.

Functions

<code>parse(source[, columns, invalid, pedantic, ...])</code>	Parses a <code>VOTABLE</code> xml file (or file-like object), and returns a <code>VOTableFile</code> object.
<code>parse_single_table(source, **kwargs)</code>	Parses a <code>VOTABLE</code> xml file (or file-like object), reading and returning only the first table.
<code>validate(source[, output, xmllint, filename])</code>	Prints a validation report for the given file.
<code>from_table(table[, table_id])</code>	Given an <code>Table</code> object, return a <code>VOTableFile</code> file structure containing just that table.
<code>is_votable(source)</code>	Reads the header of a file to determine if it is a VOTable file.
<code>writeto(table, file[, tabledata_format])</code>	Writes a <code>VOTableFile</code> to a <code>VOTABLE</code> xml file.

`parse`

`astropy.io.votable.parse` (*source*, *columns=None*, *invalid=u'exception'*, *pedantic=None*, *chunk_size=256*, *table_number=None*, *table_id=None*, *filename=None*, *unit_format=None*, *_debug_python_based_parser=False*)
Parses a `VOTABLE` xml file (or file-like object), and returns a `VOTableFile` object.

Parameters

source : str or readable file-like object

Path or file object containing a `VOTABLE` xml file.

columns : sequence of str, optional

List of field names to include in the output. The default is to include all fields.

invalid : str, optional

One of the following values:

- ‘exception’: throw an exception when an invalid value is encountered (default)
- ‘mask’: mask out invalid values

pedantic : bool, optional

When `True`, raise an error when the file violates the spec, otherwise issue a warning. Warnings may be controlled using the standard Python mechanisms. See the `warnings` module in the Python standard library for more information. When not provided, uses the configuration setting `astropy.io.votable.pedantic`, which defaults to `False`.

chunk_size : int, optional

The number of rows to read before converting to an array. Higher numbers are likely to be faster, but will consume more memory.

table_number : int, optional

The number of table in the file to read in. If `None`, all tables will be read. If a number, 0 refers to the first table in the file, and only that numbered table will be parsed and read in. Should not be used with `table_id`.

table_id : str, optional

The ID of the table in the file to read in. Should not be used with `table_number`.

filename : str, optional

A filename, URL or other identifier to use in error messages. If `filename` is `None` and `source` is a string (i.e. a path), then `source` will be used as a filename for error messages. Therefore, `filename` is only required when `source` is a file-like object.

unit_format : str, `astropy.units.format.Base` instance or `None`, optional

The unit format to use when parsing unit attributes. If a string, must be the name of a unit formatter. The built-in formats include `generic`, `fits`, `cds`, and `vounit`. A custom formatter may be provided by passing a `UnitBase` instance. If `None` (default), the unit format to use will be the one specified by the VOTable specification (which is `cds` up to version 1.2 of VOTable, and (probably) `vounit` in future versions of the spec).

Returns

votable : `VOTableFile` object

See also:

[astropy.io.votable.exceptions](#)

The exceptions this function may raise.

parse_single_table

`astropy.io.votable.parse_single_table` (*source*, ***kwargs*)

Parses a VOTABLE xml file (or file-like object), reading and returning only the first `Table` instance.

See `parse` for a description of the keyword arguments.

Returns

votable : `Table` object

validate

`astropy.io.votable.validate` (*source*, *output=None*, *xmllint=False*, *filename=None*)

Prints a validation report for the given file.

Parameters

source : str or readable file-like object

Path to a **VOTABLE** xml file.

output : writable file-like object, optional

Where to output the report. Defaults to `sys.stdout`. If `None`, the output will be returned as a string.

xmllint : bool, optional

When `True`, also send the file to `xmllint` for schema and DTD validation. Requires that `xmllint` is installed. The default is `False`. `source` must be a file on the local filesystem in order for `xmllint` to work.

filename : str, optional

A filename to use in the error messages. If not provided, one will be automatically determined from `source`.

Returns

is_valid : bool or str

Returns `True` if no warnings were found. If `output` is `None`, the return value will be a string.

from_table

`astropy.io.votable.from_table` (*table*, *table_id=None*)

Given an `Table` object, return a `VOTableFile` file structure containing just that single table.

Parameters

table : `Table` instance

table_id : str, optional

If not `None`, set the given id on the returned `Table` instance.

Returns

votable : `VOTableFile` instance

is_votable

`astropy.io.votable.is_votable` (*source*)

Reads the header of a file to determine if it is a `VOTable` file.

Parameters

source : str or readable file-like object

Path or file object containing a **VOTABLE** xml file.

Returns

is_votable : bool

Returns `True` if the given file is a `VOTable` file.

writeto

`astropy.io.votable.writeto` (*table*, *file*, *tabledata_format=None*)

Writes a `VOTableFile` to a `VOTABLE` xml file.

Parameters

table : `VOTableFile` or `Table` instance.

file : str or writable file-like object

Path or file object to write to

tabledata_format : str, optional

Override the format of the table(s) data to write. Must be one of `tabledata` (text representation), `binary` or `binary2`. By default, use the format that was specified in each `table` object as it was created or read in. See *Data serialization formats*.

Classes

`Conf` Configuration parameters for `astropy.io.votable`.

Conf

class `astropy.io.votable.Conf`

Bases: `astropy.config.ConfigNamespace`

Configuration parameters for `astropy.io.votable`.

Attributes Summary

`pedantic` When True, treat fixable violations of the `VOTable` spec as exceptions.

Attributes Documentation**pedantic**

When True, treat fixable violations of the `VOTable` spec as exceptions.

16.5.2 astropy.io.votable.tree Module**Classes**

<code>Link</code> ([ID, title, value, href, action, id, ...])	LINK elements: used to reference external documents and servers through a URI.
<code>Info</code> ([ID, name, value, id, xtype, ref, ...])	INFO elements: arbitrary key-value pairs for extensions to the standard.
<code>Values</code> (votable, field[, ID, null, ref, ...])	VALUES element: used within FIELD and PARAM elements to define the domain
<code>Field</code> (votable[, ID, name, datatype, ...])	FIELD element: describes the datatype of a particular column of data.
<code>Param</code> (votable[, ID, name, value, datatype, ...])	PARAM element: constant-valued columns in the data.
<code>CooSys</code> ([ID, equinox, epoch, system, id, ...])	COOSYS element: defines a coordinate system.
<code>FieldRef</code> (table, ref[, ucd, xtype, config, pos])	FIELDref element: used inside of GROUP elements to refer to remote FIELD elem

Continued on r

Table 16.4 – continued from previous page

<code>ParamRef(table, ref[, ucd, utype, config, pos])</code>	<code>PARAMref</code> element: used inside of <code>GROUP</code> elements to refer to remote <code>PARAM</code> elements.
<code>Group(table[, ID, name, ref, ucd, utype, ...])</code>	<code>GROUP</code> element: groups <code>FIELD</code> and <code>PARAM</code> elements.
<code>Table(votable[, ID, name, ref, ucd, utype, ...])</code>	<code>TABLE</code> element: optionally contains data.
<code>Resource([name, ID, utype, type, id, ...])</code>	<code>RESOURCE</code> element: Groups <code>TABLE</code> and <code>RESOURCE</code> elements.
<code>VOTableFile([ID, id, config, pos, version])</code>	<code>VOTABLE</code> element: represents an entire file.

Link

class `astropy.io.votable.tree.Link` (*ID=None, title=None, value=None, href=None, action=None, id=None, config=None, pos=None, **kwargs*)

Bases: `astropy.io.votable.tree.SimpleElement`, `astropy.io.votable.tree._IDProperty`

`LINK` elements: used to reference external documents and servers through a URI.

The keyword arguments correspond to setting members of the same name, documented below.

Attributes Summary

<code>content_role</code>	Defines the MIME role of the referenced object.
<code>content_type</code>	Defines the MIME content type of the referenced object.
<code>href</code>	A URI to an arbitrary protocol.

Methods Summary

<code>from_table_column(d)</code>
<code>to_table_column(column)</code>

Attributes Documentation

`content_role`

Defines the MIME role of the referenced object. Must be one of:

None, 'query', 'hints', 'doc', 'location' or 'type'

`content_type`

Defines the MIME content type of the referenced object.

`href`

A URI to an arbitrary protocol. The `vo` package only supports `http` and anonymous `ftp`.

Methods Documentation

classmethod `from_table_column` (*d*)

`to_table_column` (*column*)

Info

class `astropy.io.votable.tree.Info` (*ID=None, name=None, value=None, id=None, xtype=None, ref=None, unit=None, ucd=None, utype=None, config=None, pos=None, **extra*)

Bases: `astropy.io.votable.tree.SimpleElementWithContent`,
`astropy.io.votable.tree._IDProperty`, `astropy.io.votable.tree._XtypeProperty`,
`astropy.io.votable.tree._UtypeProperty`

INFO elements: arbitrary key-value pairs for extensions to the standard.

The keyword arguments correspond to setting members of the same name, documented below.

Attributes Summary

<code>content</code>	The content inside the INFO element.
<code>name</code>	[<i>required</i>] The key of the key-value pair.
<code>ref</code>	Refer to another INFO element by ID , defined previously in the document.
<code>unit</code>	A string specifying the units for the INFO .
<code>value</code>	[<i>required</i>] The value of the key-value pair. (Always stored

Methods Summary

<code>to_xml(w, **kwargs)</code>	For internal use.
----------------------------------	-------------------

Attributes Documentation

content

The content inside the INFO element.

name

[*required*] The key of the key-value pair.

ref

Refer to another **INFO** element by **ID**, defined previously in the document.

unit

A string specifying the **units** for the **INFO**.

value

[*required*] The value of the key-value pair. (Always stored as a string or unicode string).

Methods Documentation

`to_xml(w, **kwargs)`

For internal use. Output the element to XML.

Parameters

w : `astropy.utils.xml.writer.XMLWriter` object

An XML writer to write to.

kwargs : dict

Any configuration parameters to control the output.

Values

```
class astropy.io.votable.tree.Values(votable, field, ID=None, null=None, ref=None,
                                     type=u'legal', id=None, config=None, pos=None,
                                     **extras)
```

Bases: `astropy.io.votable.tree.Element`, `astropy.io.votable.tree._IDProperty`

VALUES element: used within **FIELD** and **PARAM** elements to define the domain of values.

The keyword arguments correspond to setting members of the same name, documented below.

Attributes Summary

<code>max</code>	The maximum value of the domain.
<code>max_inclusive</code>	When <code>True</code> , the domain includes the maximum value.
<code>min</code>	The minimum value of the domain.
<code>min_inclusive</code>	When <code>True</code> , the domain includes the minimum value.
<code>null</code>	For integral datatypes, <code>null</code> is used to define the value used for missing values.
<code>options</code>	A list of string key-value tuples defining other OPTION elements for the domain.
<code>ref</code>	Refer to another VALUES element by ID , defined previously in the document, for MIN/MAX/OPTION information.
<code>type</code>	[<i>required</i>] Defines the applicability of the domain defined

Methods Summary

<code>from_table_column(column)</code>	
<code>is_defaults()</code>	Are the settings on this VALUE element all the same as the
<code>parse(iterator, config)</code>	For internal use.
<code>to_table_column(column)</code>	
<code>to_xml(w, **kwargs)</code>	For internal use.

Attributes Documentation

max

The maximum value of the domain. See `max_inclusive`.

max_inclusive

When `True`, the domain includes the maximum value.

min

The minimum value of the domain. See `min_inclusive`.

min_inclusive

When `True`, the domain includes the minimum value.

null

For integral datatypes, `null` is used to define the value used for missing values.

options

A list of string key-value tuples defining other **OPTION** elements for the domain. All options are ignored – they are stored for round-tripping purposes only.

ref

Refer to another **VALUES** element by **ID**, defined previously in the document, for MIN/MAX/OPTION information.

type

[*required*] Defines the applicability of the domain defined by this **VALUES** element. Must be one of the following strings:

- ‘legal’: The domain of this column applies in general to this datatype. (default)
- ‘actual’: The domain of this column applies only to the data enclosed in the parent table.

Methods Documentation

from_table_column (*column*)

is_defaults ()

Are the settings on this **VALUE** element all the same as the XML defaults?

parse (*iterator, config*)

For internal use. Parse the XML content of the children of the element.

Parameters

iterator : xml iterator

An iterator over XML elements as returned by `get_xml_iterator`.

config : dict

The configuration dictionary that affects how certain elements are read.

Returns

self : Element

Returns self as a convenience.

to_table_column (*column*)

to_xml (*w, **kwargs*)

For internal use. Output the element to XML.

Parameters

w : `astropy.utils.xml.writer.XMLWriter` object

An XML writer to write to.

kwargs : dict

Any configuration parameters to control the output.

Field

class `astropy.io.votable.tree.Field` (*votable, ID=None, name=None, datatype=None, array-size=None, ucd=None, unit=None, width=None, precision=None, utype=None, ref=None, type=None, id=None, xtype=None, config=None, pos=None, **extra*)

Bases: `astropy.io.votable.tree.SimpleElement`, `astropy.io.votable.tree._IDProperty`, `astropy.io.votable.tree._NameProperty`, `astropy.io.votable.tree._XtypeProperty`, `astropy.io.votable.tree._UtypeProperty`, `astropy.io.votable.tree._UcdProperty`

FIELD element: describes the datatype of a particular column of data.

The keyword arguments correspond to setting members of the same name, documented below.

If *ID* is provided, it is used for the column name in the resulting recarray of the table. If no *ID* is provided, *name* is used instead. If neither is provided, an exception will be raised.

Attributes Summary

<code>arraysize</code>	Specifies the size of the multidimensional array if this FIELD contains more than a single value.
<code>datatype</code>	[<i>required</i>] The datatype of the column. Valid values (as
<code>links</code>	A list of Link instances used to reference more details about the meaning of the FIELD .
<code>precision</code>	Along with <code>width</code> , defines the numerical accuracy associated with the data.
<code>ref</code>	On FIELD elements, <code>ref</code> is used only for informational purposes, for example to refer to a COOSYS element.
<code>type</code>	The type attribute on FIELD elements is reserved for future extensions.
<code>unit</code>	A string specifying the units for the FIELD .
<code>values</code>	A Values instance (or <code>None</code>) defining the domain of the column.
<code>width</code>	Along with <code>precision</code> , defines the numerical accuracy associated with the data.

Methods Summary

<code>from_table_column(votable, column)</code>	Restores a Field instance from a given <code>astropy.table.Column</code> instance.
<code>parse(iterator, config)</code>	For internal use.
<code>to_table_column(column)</code>	Sets the attributes of a given <code>astropy.table.Column</code> instance to match the information.
<code>to_xml(w, **kwargs)</code>	For internal use.
<code>uniqify_names(fields)</code>	Make sure that all names and titles in a list of fields are unique, by appending numbers if necessary.

Attributes Documentation

arraysize

Specifies the size of the multidimensional array if this **FIELD** contains more than a single value.

See **multidimensional arrays**.

datatype

[*required*] The datatype of the column. Valid values (as defined by the spec) are:

‘boolean’, ‘bit’, ‘unsignedByte’, ‘short’, ‘int’, ‘long’, ‘char’, ‘unicodeChar’, ‘float’, ‘double’, ‘floatComplex’, or ‘doubleComplex’

Many VOTABLE files in the wild use ‘string’ instead of ‘char’, so that is also a valid option, though ‘string’ will always be converted to ‘char’ when writing the file back out.

links

A list of **Link** instances used to reference more details about the meaning of the **FIELD**. This is purely informational and is not used by the `astropy.io.votable` package.

precision

Along with `width`, defines the **numerical accuracy** associated with the data. These values are used to limit the precision when writing floating point values back to the XML file. Otherwise, it is purely informational – the Numpy recarray containing the data itself does not use this information.

ref

On **FIELD** elements, `ref` is used only for informational purposes, for example to refer to a **COOSYS** element.

type

The type attribute on `FIELD` elements is reserved for future extensions.

unit

A string specifying the `units` for the `FIELD`.

values

A `Values` instance (or `None`) defining the domain of the column.

width

Along with `precision`, defines the `numerical accuracy` associated with the data. These values are used to limit the precision when writing floating point values back to the XML file. Otherwise, it is purely informational – the Numpy recarray containing the data itself does not use this information.

Methods Documentation**classmethod** `from_table_column` (*votable*, *column*)

Restores a `Field` instance from a given `astropy.table.Column` instance.

parse (*iterator*, *config*)

For internal use. Parse the XML content of the children of the element.

Parameters

iterator : xml iterator

An iterator over XML elements as returned by `get_xml_iterator`.

config : dict

The configuration dictionary that affects how certain elements are read.

Returns

self : Element

Returns self as a convenience.

to_table_column (*column*)

Sets the attributes of a given `astropy.table.Column` instance to match the information in this `Field`.

to_xml (*w*, ***kwargs*)

For internal use. Output the element to XML.

Parameters

w : `astropy.utils.xml.writer.XMLWriter` object

An XML writer to write to.

kwargs : dict

Any configuration parameters to control the output.

classmethod `uniqify_names` (*fields*)

Make sure that all names and titles in a list of fields are unique, by appending numbers if necessary.

Param

```
class astropy.io.votable.tree.Param(votable, ID=None, name=None, value=None,
                                   datatype=None, arraysize=None, ucd=None, unit=None,
                                   width=None, precision=None, utype=None, type=None,
                                   id=None, config=None, pos=None, **extra)
```

Bases: `astropy.io.votable.tree.Field`

PARAM element: constant-valued columns in the data.

`Param` objects are a subclass of `Field`, and have all of its methods and members. Additionally, it defines `value`.

Attributes Summary

<code>value</code>	[<i>required</i>] The constant value of the parameter. Its type is
--------------------	--

Methods Summary

<code>to_xml(w, **kwargs)</code>	For internal use.
----------------------------------	-------------------

Attributes Documentation

value

[*required*] The constant value of the parameter. Its type is determined by the `datatype` member.

Methods Documentation

`to_xml(w, **kwargs)`

For internal use. Output the element to XML.

Parameters

w : `astropy.utils.xml.writer.XMLWriter` object

An XML writer to write to.

kwargs : dict

Any configuration parameters to control the output.

CooSys

```
class astropy.io.votable.tree.CooSys(ID=None, equinox=None, epoch=None, system=None,
                                     id=None, config=None, pos=None, **extra)
```

Bases: `astropy.io.votable.tree.SimpleElement`

COOSYS element: defines a coordinate system.

The keyword arguments correspond to setting members of the same name, documented below.

Attributes Summary

<code>ID</code>	[<i>required</i>] The XML ID of the <code>COOSYS</code> element, used for
<code>epoch</code>	Specifies the epoch of the positions.
<code>equinox</code>	A parameter required to fix the equatorial or ecliptic systems (as e.g.
<code>system</code>	Specifies the type of coordinate system.

Attributes Documentation

`ID`

[*required*] The XML ID of the `COOSYS` element, used for cross-referencing. May be `None` or a string conforming to XML ID syntax.

`epoch`

Specifies the epoch of the positions. It must be a string specifying an astronomical year.

`equinox`

A parameter required to fix the equatorial or ecliptic systems (as e.g. “J2000” as the default “eq_FK5” or “B1950” as the default “eq_FK4”).

`system`

Specifies the type of coordinate system. Valid choices are:

‘eq_FK4’, ‘eq_FK5’, ‘ICRS’, ‘ecl_FK4’, ‘ecl_FK5’, ‘galactic’, ‘supergalactic’, ‘xy’, ‘barycentric’, or ‘geo_app’

FieldRef

```
class astropy.io.votable.tree.FieldRef(table, ref, ucd=None, utype=None, config=None,
                                       pos=None, **extra)
```

Bases: `astropy.io.votable.tree.SimpleElement`, `astropy.io.votable.tree._UtypeProperty`, `astropy.io.votable.tree._UcdProperty`

`FIELDref` element: used inside of `GROUP` elements to refer to remote `FIELD` elements.

`table` is the `Table` object that this `FieldRef` is a member of.

`ref` is the ID to reference a `Field` object defined elsewhere.

Attributes Summary

<code>ref</code>	The ID of the <code>FIELD</code> that this <code>FIELDref</code> references.
------------------	--

Methods Summary

<code>get_ref()</code>	Lookup the <code>Field</code> instance that this <code>FieldRef</code> references.
------------------------	--

Attributes Documentation

`ref`

The ID of the `FIELD` that this `FIELDref` references.

Methods Documentation

`get_ref()`

Lookup the `Field` instance that this `FieldRef` references.

ParamRef

class `astropy.io.votable.tree.ParamRef` (*table, ref, ucd=None, utype=None, config=None, pos=None*)

Bases: `astropy.io.votable.tree.SimpleElement`, `astropy.io.votable.tree._UtypeProperty`, `astropy.io.votable.tree._UcdProperty`

PARAMref element: used inside of **GROUP** elements to refer to remote **PARAM** elements.

The keyword arguments correspond to setting members of the same name, documented below.

It contains the following publicly-accessible members:

ref: An XML ID referring to a `<PARAM>` element.

Attributes Summary

`ref` The ID of the **PARAM** that this **PARAMref** references.

Methods Summary

`get_ref()` Lookup the `Param` instance that this `:class:PARAMref` references.

Attributes Documentation

ref

The ID of the **PARAM** that this **PARAMref** references.

Methods Documentation

`get_ref()`

Lookup the `Param` instance that this `:class:PARAMref` references.

Group

class `astropy.io.votable.tree.Group` (*table, ID=None, name=None, ref=None, ucd=None, utype=None, id=None, config=None, pos=None, **extra*)

Bases: `astropy.io.votable.tree.Element`, `astropy.io.votable.tree._IDProperty`, `astropy.io.votable.tree._NameProperty`, `astropy.io.votable.tree._UtypeProperty`, `astropy.io.votable.tree._UcdProperty`, `astropy.io.votable.tree._DescriptionProperty`

GROUP element: groups **FIELD** and **PARAM** elements.

This information is currently ignored by the `vo` package—that is the columns in the recarray are always flat—but the grouping information is stored so that it can be written out again to the XML file.

The keyword arguments correspond to setting members of the same name, documented below.

Attributes Summary

<code>entries</code>	[read-only] A list of members of the <code>GROUP</code> . This list may
<code>ref</code>	Currently ignored, as it's not clear from the spec how this is meant to work.

Methods Summary

<code>iter_fields_and_params()</code>	Recursively iterate over all <code>Param</code> elements in this <code>Group</code> .
<code>iter_groups()</code>	Recursively iterate over all sub- <code>Group</code> instances in this <code>Group</code> .
<code>parse(iterator, config)</code>	For internal use.
<code>to_xml(w, **kwargs)</code>	For internal use.

Attributes Documentation

`entries`

[read-only] A list of members of the `GROUP`. This list may only contain objects of type `Param`, `Group`, `ParamRef` and `FieldRef`.

`ref`

Currently ignored, as it's not clear from the spec how this is meant to work.

Methods Documentation

`iter_fields_and_params()`

Recursively iterate over all `Param` elements in this `Group`.

`iter_groups()`

Recursively iterate over all sub-`Group` instances in this `Group`.

`parse(iterator, config)`

For internal use. Parse the XML content of the children of the element.

Parameters

iterator : xml iterator

An iterator over XML elements as returned by `get_xml_iterator`.

config : dict

The configuration dictionary that affects how certain elements are read.

Returns

self : Element

Returns self as a convenience.

`to_xml(w, **kwargs)`

For internal use. Output the element to XML.

Parameters

w : `astropy.utils.xml.writer.XMLWriter` object

An XML writer to write to.

kwargs : dict

Any configuration parameters to control the output.

Table

class `astropy.io.votable.tree.Table` (*votable*, *ID=None*, *name=None*, *ref=None*, *ucd=None*, *utype=None*, *nrows=None*, *id=None*, *config=None*, *pos=None*, ***extra*)

Bases: `astropy.io.votable.tree.Element`, `astropy.io.votable.tree._IDProperty`, `astropy.io.votable.tree._NameProperty`, `astropy.io.votable.tree._UcdProperty`, `astropy.io.votable.tree._DescriptionProperty`

TABLE element: optionally contains data.

It contains the following publicly-accessible and mutable attribute:

array: A Numpy masked array of the data itself, where each row is a row of votable data, and columns are named and typed based on the <FIELD> elements of the table. The mask is parallel to the data array, except for variable-length fields. For those fields, the numpy array's column type is "object" ("O"), and another masked array is stored there.

If the Table contains no data, (for example, its enclosing `Resource` has `type == 'meta'`) *array* will have zero-length.

The keyword arguments correspond to setting members of the same name, documented below.

Attributes Summary

<code>fields</code>	A list of <code>Field</code> objects describing the types of each of the data columns.
<code>format</code>	[<i>required</i>] The serialization format of the table. Must be
<code>groups</code>	A list of <code>Group</code> objects describing how the columns and parameters are grouped.
<code>infos</code>	A list of <code>Info</code> objects for the table.
<code>links</code>	A list of <code>Link</code> objects (pointers to other documents or servers through a URI) for the table.
<code>nrows</code>	[<i>immutable</i>] The number of rows in the table, as specified in
<code>params</code>	A list of parameters (constant-valued columns) for the table.
<code>ref</code>	

Methods Summary

<code>create_arrays([nrows, config])</code>	Create a new array to hold the data based on the current set of fields, and store them there.
<code>from_table(votable, table)</code>	Create a <code>Table</code> instance from a given <code>astropy.table.Table</code> instance.
<code>get_field_by_id(ref[, before])</code>	Looks up a FIELD or PARAM element by the given ID.
<code>get_field_by_id_or_name(ref[, before])</code>	Looks up a FIELD or PARAM element by the given ID or name.
<code>get_fields_by_utype(ref[, before])</code>	Looks up a FIELD or PARAM element by the given utype and returns an iterator over them.
<code>get_group_by_id(ref[, before])</code>	Looks up a GROUP element by the given ID.
<code>get_groups_by_utype(ref[, before])</code>	Looks up a GROUP element by the given utype and returns an iterator emitting all of them.
<code>is_empty()</code>	Returns True if this table doesn't contain any real data because it was skipped over.
<code>iter_fields_and_params()</code>	Recursively iterate over all FIELD and PARAM elements in the TABLE.
<code>iter_groups()</code>	Recursively iterate over all GROUP elements in the TABLE.
<code>parse(iterator, config)</code>	For internal use.
<code>to_table([use_names_over_ids])</code>	Convert this VO Table to an <code>astropy.table.Table</code> instance.

`to_xml(w, **kwargs)`

For internal use.

Attributes Documentation

fields

A list of `Field` objects describing the types of each of the data columns.

format

[*required*] The serialization format of the table. Must be one of:

‘tabledata’ (`TABLEDATA`), ‘binary’ (`BINARY`), ‘binary2’ (`BINARY2`) ‘fits’ (`FITS`).

Note that the ‘fits’ format, since it requires an external file, can not be written out. Any file read in with ‘fits’ format will be read out, by default, in ‘tabledata’ format.

See *Data serialization formats*.

groups

A list of `Group` objects describing how the columns and parameters are grouped. Currently this information is only kept around for round-tripping and informational purposes.

infos

A list of `Info` objects for the table. Allows for post-operational diagnostics.

links

A list of `Link` objects (pointers to other documents or servers through a URI) for the table.

nrows

[*immutable*] The number of rows in the table, as specified in the XML file.

params

A list of parameters (constant-valued columns) for the table. Must contain only `Param` objects.

ref

Methods Documentation

create_arrays (*nrows=0, config=None*)

Create a new array to hold the data based on the current set of fields, and store them in the *array* and member variable. Any data in the existing array will be lost.

nrows, if provided, is the number of rows to allocate.

classmethod from_table (*votable, table*)

Create a `Table` instance from a given `astropy.table.Table` instance.

get_field_by_id (*ref, before=None*)

Looks up a FIELD or PARAM element by the given ID.

get_field_by_id_or_name (*ref, before=None*)

Looks up a FIELD or PARAM element by the given ID or name.

get_fields_by_ctype (*ref, before=None*)

Looks up a FIELD or PARAM element by the given ctype and returns an iterator emitting all matches.

get_group_by_id (*ref, before=None*)

Looks up a GROUP element by the given ID. Used by the group’s “ref” attribute

get_groups_by_utype (*ref*, *before=None*)

Looks up a GROUP element by the given utype and returns an iterator emitting all matches.

is_empty ()

Returns True if this table doesn't contain any real data because it was skipped over by the parser (through use of the `table_number` kwarg).

iter_fields_and_params ()

Recursively iterate over all FIELD and PARAM elements in the TABLE.

iter_groups ()

Recursively iterate over all GROUP elements in the TABLE.

parse (*iterator*, *config*)

For internal use. Parse the XML content of the children of the element.

Parameters

iterator : xml iterator

An iterator over XML elements as returned by `get_xml_iterator`.

config : dict

The configuration dictionary that affects how certain elements are read.

Returns

self : Element

Returns self as a convenience.

to_table (*use_names_over_ids=False*)

Convert this VO Table to an `astropy.table.Table` instance.

Parameters

use_names_over_ids : bool, optional

When `True` use the name attributes of columns as the names of columns in the `astropy.table.Table` instance. Since names are not guaranteed to be unique, this may cause some columns to be renamed by appending numbers to the end. Otherwise (default), use the ID attributes as the column names.

.. warning::

Variable-length array fields may not be restored identically when round-tripping through the `astropy.table.Table` instance.

to_xml (*w*, ***kwargs*)

For internal use. Output the element to XML.

Parameters

w : `astropy.utils.xml.writer.XMLWriter` object

An XML writer to write to.

kwargs : dict

Any configuration parameters to control the output.

Resource

```
class astropy.io.votable.tree.Resource (name=None, ID=None, utype=None, type='u'results',
                                       id=None, config=None, pos=None, **kwargs)
```

Bases: `astropy.io.votable.tree.Element`, `astropy.io.votable.tree._IDProperty`,

`astropy.io.votable.tree._NameProperty`, `astropy.io.votable.tree._UtypeProperty`,
`astropy.io.votable.tree._DescriptionProperty`

RESOURCE element: Groups **TABLE** and **RESOURCE** elements.

The keyword arguments correspond to setting members of the same name, documented below.

Attributes Summary

<code>coordinate_systems</code>	A list of coordinate system definitions (COOSYS elements) for the RESOURCE .
<code>extra_attributes</code>	A dictionary of string keys to string values containing any extra attributes of the RESOURCE element that are not defined in the specification. (The specification explicitly allows for extra attributes here, but nowhere else.)
<code>infos</code>	A list of informational parameters (key-value pairs) for the resource. Must only contain Info objects.
<code>links</code>	A list of links (pointers to other documents or servers through a URI) for the resource. Must contain only Link objects.
<code>params</code>	A list of parameters (constant-valued columns) for the resource. Must contain only Param objects.
<code>resources</code>	A list of nested resources inside this resource. Must contain only Resource objects.
<code>tables</code>	A list of tables in the resource. Must contain only Table objects.
<code>type</code>	[<i>required</i>] The type of the resource. Must be either:

Methods Summary

<code>iter_coosys()</code>	Recursively iterates over all the COOSYS elements in the resource and nested resources.
<code>iter_fields_and_params()</code>	Recursively iterates over all FIELD and PARAM elements in the resource, its tables and nested resources.
<code>iter_tables()</code>	Recursively iterates over all tables in the resource and nested resources.
<code>parse(votable, iterator, config)</code>	For internal use.
<code>to_xml(w, **kwargs)</code>	For internal use.

Attributes Documentation

coordinate_systems

A list of coordinate system definitions (**COOSYS** elements) for the **RESOURCE**. Must contain only **CooSys** objects.

extra_attributes

A dictionary of string keys to string values containing any extra attributes of the **RESOURCE** element that are not defined in the specification. (The specification explicitly allows for extra attributes here, but nowhere else.)

infos

A list of informational parameters (key-value pairs) for the resource. Must only contain **Info** objects.

links

A list of links (pointers to other documents or servers through a URI) for the resource. Must contain only **Link** objects.

params

A list of parameters (constant-valued columns) for the resource. Must contain only **Param** objects.

resources

A list of nested resources inside this resource. Must contain only **Resource** objects.

tables

A list of tables in the resource. Must contain only **Table** objects.

type

[*required*] The type of the resource. Must be either:

- ‘results’: This resource contains actual result values (default)
- ‘meta’: This resource contains only datatype descriptions (**FIELD** elements), but no actual data.

Methods Documentation**iter_coosys ()**

Recursively iterates over all the **COOSYS** elements in the resource and nested resources.

iter_fields_and_params ()

Recursively iterates over all **FIELD** and **PARAM** elements in the resource, its tables and nested resources.

iter_tables ()

Recursively iterates over all tables in the resource and nested resources.

parse (votable, iterator, config)

For internal use. Parse the XML content of the children of the element.

Parameters

iterator : xml iterator

An iterator over XML elements as returned by `get_xml_iterator`.

config : dict

The configuration dictionary that affects how certain elements are read.

Returns

self : Element

Returns self as a convenience.

to_xml (w, **kwargs)

For internal use. Output the element to XML.

Parameters

w : `astropy.utils.xml.writer.XMLWriter` object

An XML writer to write to.

kwargs : dict

Any configuration parameters to control the output.

VOTableFile

class `astropy.io.votable.tree.VOTableFile (ID=None, id=None, config=None, pos=None, version=u'1.2')`

Bases: `astropy.io.votable.tree.Element`, `astropy.io.votable.tree._IDProperty`, `astropy.io.votable.tree._DescriptionProperty`

VOTABLE element: represents an entire file.

The keyword arguments correspond to setting members of the same name, documented below.

version is settable at construction time only, since conformance tests for building the rest of the structure depend on it.

Attributes Summary

<code>coordinate_systems</code>	A list of coordinate system descriptions for the file.
<code>groups</code>	A list of groups, in the order they appear in the file.
<code>infos</code>	A list of informational parameters (key-value pairs) for the entire file.
<code>params</code>	A list of parameters (constant-valued columns) that apply to the entire file.
<code>resources</code>	A list of resources, in the order they appear in the file.
<code>version</code>	The version of the VOTable specification that the file uses.

Methods Summary

<code>from_table(table[, table_id])</code>	Create a <code>VOTableFile</code> instance from a given <code>astropy.table.Table</code> instance.
<code>get_coosys_by_id(ref[, before])</code>	Looks up a <code>COOSYS</code> element by the given <code>ID</code> .
<code>get_field_by_id(ref[, before])</code>	Looks up a <code>FIELD</code> element by the given <code>ID</code> .
<code>get_field_by_id_or_name(ref[, before])</code>	Looks up a <code>FIELD</code> element by the given <code>ID</code> or name.
<code>get_fields_by_ctype(ref[, before])</code>	Looks up a <code>FIELD</code> element by the given <code>ctype</code> and returns an iterator emitting all matches.
<code>get_first_table()</code>	Often, you know there is only one table in the file, and that's all you need.
<code>get_group_by_id(ref[, before])</code>	Looks up a <code>GROUP</code> element by the given <code>ID</code> .
<code>get_groups_by_ctype(ref[, before])</code>	Looks up a <code>GROUP</code> element by the given <code>ctype</code> and returns an iterator emitting all matches.
<code>get_table_by_id(ref[, before])</code>	Looks up a <code>TABLE</code> element by the given <code>ID</code> .
<code>get_table_by_index(idx)</code>	Get a table by its ordinal position in the file.
<code>get_tables_by_ctype(ref[, before])</code>	Looks up a <code>TABLE</code> element by the given <code>ctype</code> , and returns an iterator emitting all matches.
<code>get_values_by_id(ref[, before])</code>	Looks up a <code>VALUES</code> element by the given <code>ID</code> .
<code>iter_coosys()</code>	Recursively iterate over all <code>COOSYS</code> elements in the <code>VOTABLE</code> file.
<code>iter_fields_and_params()</code>	Recursively iterate over all <code>FIELD</code> and <code>PARAM</code> elements in the <code>VOTABLE</code> file.
<code>iter_groups()</code>	Recursively iterate over all <code>GROUP</code> elements in the <code>VOTABLE</code> file.
<code>iter_tables()</code>	Iterates over all tables in the VOTable file in a “flat” way, ignoring the nesting of tables.
<code>iter_values()</code>	Recursively iterate over all <code>VALUES</code> elements in the <code>VOTABLE</code> file.
<code>parse(iterator, config)</code>	For internal use.
<code>set_all_tables_format(format)</code>	Set the output storage format of all tables in the file.
<code>to_xml(fd[, write_null_values, compressed, ...])</code>	Write to an XML file.

Attributes Documentation

`coordinate_systems`

A list of coordinate system descriptions for the file. Must contain only `CooSys` objects.

`groups`

A list of groups, in the order they appear in the file. Only supported as a child of the `VOTABLE` element in VOTable 1.2 or later.

`infos`

A list of informational parameters (key-value pairs) for the entire file. Must only contain `Info` objects.

`params`

A list of parameters (constant-valued columns) that apply to the entire file. Must contain only `Param` objects.

`resources`

A list of resources, in the order they appear in the file. Must only contain `Resource` objects.

`version`

The version of the VOTable specification that the file uses.

Methods Documentation

classmethod `from_table` (*table*, *table_id=None*)

Create a `VOTableFile` instance from a given `astropy.table.Table` instance.

Parameters

table_id : str, optional

Set the given ID attribute on the returned Table instance.

get_coosys_by_id (*ref*, *before=None*)

Looks up a `COOSYS` element by the given ID.

get_field_by_id (*ref*, *before=None*)

Looks up a `FIELD` element by the given ID. Used by the field’s “ref” attribute.

get_field_by_id_or_name (*ref*, *before=None*)

Looks up a `FIELD` element by the given ID or name.

get_fields_by_utype (*ref*, *before=None*)

Looks up a `FIELD` element by the given utype and returns an iterator emitting all matches.

get_first_table ()

Often, you know there is only one table in the file, and that’s all you need. This method returns that first table.

get_group_by_id (*ref*, *before=None*)

Looks up a `GROUP` element by the given ID. Used by the group’s “ref” attribute

get_groups_by_utype (*ref*, *before=None*)

Looks up a `GROUP` element by the given utype and returns an iterator emitting all matches.

get_table_by_id (*ref*, *before=None*)

Looks up a `TABLE` element by the given ID. Used by the table “ref” attribute.

get_table_by_index (*idx*)

Get a table by its ordinal position in the file.

get_tables_by_utype (*ref*, *before=None*)

Looks up a `TABLE` element by the given utype, and returns an iterator emitting all matches.

get_values_by_id (*ref*, *before=None*)

Looks up a `VALUES` element by the given ID. Used by the values “ref” attribute.

iter_coosys ()

Recursively iterate over all `COOSYS` elements in the `VOTABLE` file.

iter_fields_and_params ()

Recursively iterate over all `FIELD` and `PARAM` elements in the `VOTABLE` file.

iter_groups ()

Recursively iterate over all `GROUP` elements in the `VOTABLE` file.

iter_tables ()

Iterates over all tables in the `VOTable` file in a “flat” way, ignoring the nesting of resources etc.

iter_values ()

Recursively iterate over all `VALUES` elements in the `VOTABLE` file.

parse (*iterator*, *config*)

For internal use. Parse the XML content of the children of the element.

Parameters

iterator : xml iterator

An iterator over XML elements as returned by `get_xml_iterator`.

config : dict

The configuration dictionary that affects how certain elements are read.

Returns

self : Element

Returns self as a convenience.

set_all_tables_format (*format*)

Set the output storage format of all tables in the file.

to_xml (*fd*, *write_null_values=False*, *compressed=False*, *tabledata_format=None*, *_debug_python_based_parser=False*, *_astropy_version=None*)

Write to an XML file.

Parameters

fd : str path or writable file-like object

Where to write the file.

write_null_values : bool, optional

Deprecated and retained for backward compatibility. When `write_null_values` was `False`, invalid VOTable files could be generated, so the option has just been removed entirely.

compressed : bool, optional

When `True`, write to a gzip-compressed file. (Default: `False`)

tabledata_format : str, optional

Override the format of the table(s) data to write. Must be one of `tabledata` (text representation), `binary` or `binary2`. By default, use the format that was specified in each `Table` object as it was created or read in. See *Data serialization formats*.

16.5.3 astropy.io.votable.converters Module

This module handles the conversion of various VOTABLE datatypes to/from `TABLEDATA` and `BINARY` formats.

Functions

`get_converter`(*field*[, *config*, *pos*])

Get an appropriate converter instance for a given field.

`table_column_to_votable_datatype`(*column*)

Given a `astropy.table.Column` instance, returns the attributes needed

`get_converter`

`astropy.io.votable.converters.get_converter` (*field*, *config=None*, *pos=None*)

Get an appropriate converter instance for a given field.

Parameters

field : `astropy.io.votable.tree.Field`

config : dict, optional

Parser configuration dictionary

pos : tuple

Position in the input XML file. Used for error messages.

Returns

converter : `astropy.io.votable.converters.Converter`

`table_column_to_votable_datatype`

`astropy.io.votable.converters.table_column_to_votable_datatype` (*column*)

Given a `astropy.table.Column` instance, returns the attributes necessary to create a VOTable FIELD element that corresponds to the type of the column.

This necessarily must perform some heuristics to determine the type of variable length arrays fields, since they are not directly supported by Numpy.

If the column has dtype of “object”, it performs the following tests:

- If all elements are byte or unicode strings, it creates a variable-length byte or unicode field, respectively.
- If all elements are numpy arrays of the same dtype and with a consistent shape in all but the first dimension, it creates a variable length array of fixed sized arrays. If the dtypes match, but the shapes do not, a variable length array is created.

If the dtype of the input is not understood, it sets the data type to the most inclusive: a variable length unicodeChar array.

Parameters

column : `astropy.table.Column` instance

Returns

attributes : dict

A dict containing ‘datatype’ and ‘arraysize’ keys that can be set on a VOTable FIELD element.

Classes

`Converter`(*field*[, *config*, *pos*]) The base class for all converters.

Converter

class `astropy.io.votable.converters.Converter` (*field*, *config=None*, *pos=None*)

Bases: `object`

The base class for all converters. Each subclass handles converting a specific VOTABLE data type to/from the `TABLEDATA` and `BINARY` on-disk representations.

Parameters

field : `Field`

object describing the datatype

config : dict

The parser configuration dictionary

pos : tuple

The position in the XML file where the FIELD object was found. Used for error messages.

Methods Summary

<code>binoutput(value, mask)</code>	Convert the object <i>value</i> in the native in-memory datatype to a string of bytes suitable for serialization in the BINARY format.
<code>binparse(read)</code>	Reads some number of bytes from the BINARY format representation by calling the function <i>read</i> , and returns the native in-memory object representation for the datatype handled by <i>self</i> .
<code>output(value, mask)</code>	Convert the object <i>value</i> (in the native in-memory datatype) to a unicode string suitable for serializing in the TABLEDATA format.
<code>parse(value[, config, pos])</code>	Convert the string <i>value</i> from the TABLEDATA format into an object with the correct native datatype.
<code>parse_scalar(value[, config, pos])</code>	Parse a single scalar of the underlying type of the converter.
<code>supports_empty_values(config)</code>	Returns True when the field can be completely empty.

Methods Documentation

binoutput (*value, mask*)

Convert the object *value* in the native in-memory datatype to a string of bytes suitable for serialization in the **BINARY** format.

Parameters

value : native type corresponding to this converter

The value

mask : bool

If **True**, will return the string representation of a masked value.

Returns

bytes : byte string

The binary representation of the value, suitable for serialization in the **BINARY** format.

binparse (*read*)

Reads some number of bytes from the **BINARY** format representation by calling the function *read*, and returns the native in-memory object representation for the datatype handled by *self*.

Parameters

read : function

A function that given a number of bytes, returns a byte string.

Returns

native : tuple (value, mask)

The value as a Numpy array or scalar, and *mask* is **True** if the value is missing.

output (*value, mask*)

Convert the object *value* (in the native in-memory datatype) to a unicode string suitable for serializing in the **TABLEDATA** format.

Parameters

value : native type corresponding to this converter

The value

mask : bool

If **True**, will return the string representation of a masked value.

Returns**tabledata_repr** : unicode**parse** (*value*, *config=None*, *pos=None*)Convert the string *value* from the TABLEDATA format into an object with the correct native in-memory datatype and mask flag.**Parameters****value** : str

value in TABLEDATA format

Returns**native** : tuple (value, mask)The value as a Numpy array or scalar, and *mask* is True if the value is missing.**parse_scalar** (*value*, *config=None*, *pos=None*)

Parse a single scalar of the underlying type of the converter. For non-array converters, this is equivalent to parse. For array converters, this is used to parse a single element of the array.

Parameters**value** : str

value in TABLEDATA format

Returns**native** : tuple (value, mask)The value as a Numpy array or scalar, and *mask* is True if the value is missing.**supports_empty_values** (*config*)

Returns True when the field can be completely empty.

16.5.4 astropy.io.votable.ucd Module

This file contains routines to verify the correctness of UCD strings.

Functions

<code>parse_ucd(ucd[, ...])</code>	Parse the UCD into its component parts.
<code>check_ucd(ucd[, ...])</code>	Returns False if <i>ucd</i> is not a valid unified content descriptor.

parse_ucd

`astropy.io.votable.ucd.parse_ucd(ucd, check_controlled_vocabulary=False, has_colon=False)`

Parse the UCD into its component parts.

Parameters**ucd** : str

The UCD string

check_controlled_vocabulary : bool, optionalIf `True`, then each word in the UCD will be verified against the UCD1+ controlled vocabulary, (as required by the VOTable specification version 1.2), otherwise not.**has_colon** : bool, optional

If `True`, the UCD may contain a colon (as defined in earlier versions of the standard).

Returns

parts : list

The result is a list of tuples of the form:

(namespace, word)

If no namespace was explicitly specified, *namespace* will be returned as `'ivoa'` (i.e., the default namespace).

Raises

ValueError : *ucd* is invalid

check_ucd

`astropy.io.votable.ucd.check_ucd(ucd, check_controlled_vocabulary=False, has_colon=False)`
Returns `False` if *ucd* is not a valid unified content descriptor.

Parameters

ucd : str

The UCD string

check_controlled_vocabulary : bool, optional

If `True`, then each word in the UCD will be verified against the UCD1+ controlled vocabulary, (as required by the VOTable specification version 1.2), otherwise not.

has_colon : bool, optional

If `True`, the UCD may contain a colon (as defined in earlier versions of the standard).

Returns

valid : bool

16.5.5 astropy.io.votable.util Module

Various utilities and cookbook-like things.

Functions

<code>convert_to_writable_filelike(*args, **kwargs)</code>	Returns a writable file-like object suitable for streaming output.
<code>coerce_range_list_param(p[, frames, numeric])</code>	Coerces and/or verifies the object <i>p</i> into a valid range-list-format parameter.

convert_to_writable_filelike

`astropy.io.votable.util.convert_to_writable_filelike(*args, **kwargs)`
Returns a writable file-like object suitable for streaming output.

Parameters

fd : file path string or writable file-like object

May be:

- a file path, in which case it is opened, and the file object is returned.

- an object with a `:meth:write` method, in which case that object is returned.

compressed : bool, optional

If `True`, create a gzip-compressed file. (Default is `False`).

Returns

fd : writable file-like object

coerce_range_list_param

`astropy.io.votable.util.coerce_range_list_param(p, frames=None, numeric=True)`

Coerces and/or verifies the object *p* into a valid range-list-format parameter.

As defined in [Section 8.7.2 of Simple Spectral Access Protocol](#).

Parameters

p : str or sequence

May be a string as passed verbatim to the service expecting a range-list, or a sequence.

If a sequence, each item must be either:

- a numeric value
- a named value, such as, for example, 'J' for named spectrum (if the *numeric* kwarg is `False`)
- a 2-tuple indicating a range
- the last item may be a string indicating the frame of reference

frames : sequence of str, optional

A sequence of acceptable frame of reference keywords. If not provided, the default set in `set_reference_frames` will be used.

numeric : bool, optional

TODO

Returns

parts : tuple

The result is a tuple:

- a string suitable for passing to a service as a range-list argument
- an integer counting the number of elements

16.5.6 astropy.io.votable.validator Module

Validates a large collection of web-accessible VOTable files, and generates a report as a directory tree of HTML files.

Functions

`make_validation_report(urls, destdir, ...)` Validates a large collection of web-accessible VOTable files.

make_validation_report

```
astropy.io.votable.validator.make_validation_report (urls=None, dest-
                                                    dir=u'astropy.io.votable.validator.results',
                                                    multiprocessing=True,
                                                    stilts=None)
```

Validates a large collection of web-accessible VOTable files.

Generates a report as a directory tree of HTML files.

Parameters

urls : list of strings, optional

If provided, is a list of HTTP urls to download VOTable files from. If not provided, a built-in set of ~22,000 urls compiled by HEASARC will be used.

destdir : path, optional

The directory to write the report to. By default, this is a directory called 'results' in the current directory. If the directory does not exist, it will be created.

multiprocess : bool, optional

If `True` (default), perform validations in parallel using all of the cores on this machine.

stilts : path, optional

To perform validation with `votlint` from the the Java-based `STILTS` VOTable parser, in addition to `astropy.io.votable`, set this to the path of the 'stilts.jar' file. `java` on the system shell path will be used to run it.

Notes

Downloads of each given URL will be performed only once and cached locally in `destdir`. To refresh the cache, remove `destdir` first.

16.5.7 astropy.io.votable.xmlutil Module

Various XML-related utilities

Functions

<code>check_id(ID[, name, config, pos])</code>	Raises a <code>VOTableSpecError</code> if <code>ID</code> is not a valid XML ID.
<code>fix_id(ID[, config, pos])</code>	Given an arbitrary string, create one that can be used as an xml id.
<code>check_token(token, attr_name[, config, pos])</code>	Raises a <code>ValueError</code> if <code>token</code> is not a valid XML token.
<code>check_mime_content_type(content_type[, ...])</code>	Raises a <code>VOTableSpecError</code> if <code>content_type</code> is not a valid MIME content
<code>check_anyuri(uri[, config, pos])</code>	Raises a <code>VOTableSpecError</code> if <code>uri</code> is not a valid URI.
<code>validate_schema(filename[, version])</code>	Validates the given file against the appropriate VOTable schema.

check_id

```
astropy.io.votable.xmlutil.check_id (ID, name=u'ID', config=None, pos=None)
```

Raises a `VOTableSpecError` if `ID` is not a valid XML ID.

`name` is the name of the attribute being checked (used only for error messages).

fix_id

`astropy.io.votable.xmlutil.fix_id` (*ID*, *config=None*, *pos=None*)

Given an arbitrary string, create one that can be used as an xml id.

This is rather simplistic at the moment, since it just replaces non-valid characters with underscores.

check_token

`astropy.io.votable.xmlutil.check_token` (*token*, *attr_name*, *config=None*, *pos=None*)

Raises a `ValueError` if *token* is not a valid XML token.

As defined by XML Schema Part 2.

check_mime_content_type

`astropy.io.votable.xmlutil.check_mime_content_type` (*content_type*, *config=None*,
pos=None)

Raises a `VOTableSpecError` if *content_type* is not a valid MIME content type.

As defined by RFC 2045 (syntactically, at least).

check_anyuri

`astropy.io.votable.xmlutil.check_anyuri` (*uri*, *config=None*, *pos=None*)

Raises a `VOTableSpecError` if *uri* is not a valid URI.

As defined in RFC 2396.

validate_schema

`astropy.io.votable.xmlutil.validate_schema` (*filename*, *version=u'1.1'*)

Validates the given file against the appropriate VOTable schema.

Parameters

filename : str

The path to the XML file to validate

version : str, optional

The VOTABLE version to check, which must be a string “1.0”, “1.1”, “1.2” or “1.3”. If it is not one of these, version “1.1” is assumed.

For version “1.0”, it is checked against a DTD, since that version did not have an XML Schema.

Returns

returncode, stdout, stderr : int, str, str

Returns the returncode from xmllint and the stdout and stderr as strings

16.5.8 `astropy.io.votable.exceptions` Module

`astropy.io.votable.exceptions`

Contents

- `astropy.io.votable.exceptions`
 - Warnings
 - * W01: Array uses commas rather than whitespace
 - * W02: x attribute ‘y’ is invalid. Must be a standard XML id
 - * W03: Implicitly generating an ID from a name ‘x’ -> ‘y’
 - * W04: content-type ‘x’ must be a valid MIME content type
 - * W05: ‘x’ is not a valid URI
 - * W06: Invalid UCD ‘x’: explanation
 - * W07: Invalid astroYear in x: ‘y’
 - * W08: ‘x’ must be a str or unicode object
 - * W09: ID attribute not capitalized
 - * W10: Unknown tag ‘x’. Ignoring
 - * W11: The `gref` attribute on `LINK` is deprecated in `VOTable 1.1`
 - * W12: ‘x’ element must have at least one of ‘ID’ or ‘name’ attributes
 - * W13: ‘x’ is not a valid `VOTable` datatype, should be ‘y’
 - * W15: x element missing required ‘name’ attribute
 - * W17: x element contains more than one `DESCRIPTION` element
 - * W18: `TABLE` specified `nrows=x`, but table contains y rows
 - * W19: The fields defined in the `VOTable` do not match those in the embedded FITS file
 - * W20: No version number specified in file. Assuming 1.1
 - * W21: `vo.table` is designed for `VOTable` version 1.1, 1.2 and 1.3, but this file is x
 - * W22: The `DEFINITIONS` element is deprecated in `VOTable 1.1`. Ignoring
 - * W23: Unable to update service information for ‘x’
 - * W24: The VO catalog database is for a later version of `vo.table`
 - * W25: ‘service’ failed with: ...
 - * W26: ‘child’ inside ‘parent’ added in `VOTable X.X`
 - * W27: `COOSYS` deprecated in `VOTable 1.2`
 - * W28: ‘attribute’ on ‘element’ added in `VOTable X.X`
 - * W29: Version specified in non-standard form ‘v1.0’
 - * W30: Invalid literal for float ‘x’. Treating as empty.
 - * W31: NaN given in an integral field without a specified null value
 - * W32: Duplicate ID ‘x’ renamed to ‘x_2’ to ensure uniqueness
 - * W33: Column name ‘x’ renamed to ‘x_2’ to ensure uniqueness
 - * W34: ‘x’ is an invalid token for attribute ‘y’
 - * W35: ‘x’ attribute required for `INFO` elements
 - * W36: null value ‘x’ does not match field datatype, setting to 0
 - * W37: Unsupported data format ‘x’
 - * W38: Inline binary data must be base64 encoded, got ‘x’
 - * W39: Bit values can not be masked
 - * W40: ‘cprojection’ datatype repaired
 - * W41: An XML namespace is specified, but is incorrect. Expected ‘x’, got ‘y’
 - * W42: No XML namespace specified
 - * W43: element `ref=‘x’` which has not already been defined
 - * W44: `VALUES` element with `ref` attribute has content (‘element’)
 - * W45: content-role attribute ‘x’ invalid
 - * W46: char or unicode value is too long for specified length of x
 - * W47: Missing `arraysize` indicates length 1
 - * W48: Unknown attribute ‘attribute’ on element
 - * W49: Empty cell illegal for integer fields.
 - * W50: Invalid unit string ‘x’
 - * W51: Value ‘x’ is out of range for a n-bit integer field
 - * W52: The `BINARY2` format was introduced in `VOTable 1.3`, but this file is declared as version ‘1.2’
 - * W53: `VOTABLE` element must contain at least one `RESOURCE` element.

16.5. Reference/API

- * E01: Invalid size specifier ‘x’ for a char/unicode field (in field ‘y’)
- * E02: Incorrect number of elements in array. Expected multiple of x, got y
- * E03: ‘x’ does not parse as a complex number
- * E04: Invalid bit value ‘x’

Warnings

Note: Most of the following warnings indicate violations of the VOTable specification. They should be reported to the authors of the tools that produced the VOTable file.

To control the warnings emitted, use the standard Python `warnings` module. Most of these are of the type `VOTableSpecWarning`.

W01: Array uses commas rather than whitespace The VOTable spec states:

If a cell contains an array or complex number, it should be encoded as multiple numbers separated by whitespace.

Many VOTable files in the wild use commas as a separator instead, and `vo.table` supports this convention when not in *Pedantic mode*.

`vo.table` always outputs files using only spaces, regardless of how they were input.

References: [1.1](#), [1.2](#)

W02: x attribute ‘y’ is invalid. Must be a standard XML id XML ids must match the following regular expression:

```
^[A-Za-z_][A-Za-z0-9_\.\-]*$
```

The VOTable 1.1 says the following:

According to the XML standard, the attribute `ID` is a string beginning with a letter or underscore (`_`), followed by a sequence of letters, digits, or any of the punctuation characters `.` (dot), `-` (dash), `_` (underscore), or `:` (colon).

However, this is in conflict with the XML standard, which says colons may not be used. VOTable 1.1’s own schema does not allow a colon here. Therefore, `vo.table` disallows the colon.

VOTable 1.2 corrects this error in the specification.

References: [1.1](#), [XML Names](#)

W03: Implicitly generating an ID from a name ‘x’ -> ‘y’ The VOTable 1.1 spec says the following about `name` vs. `ID` on `FIELD` and `VALUE` elements:

`ID` and `name` attributes have a different role in VOTable: the `ID` is meant as a *unique identifier* of an element seen as a VOTable component, while the `name` is meant for presentation purposes, and need not to be unique throughout the VOTable document. The `ID` attribute is therefore required in the elements which have to be referenced, but in principle any element may have an `ID` attribute. ... In summary, the `ID` is different from the `name` attribute in that (a) the `ID` attribute is made from a restricted character set, and must be unique throughout a VOTable document whereas names are standard XML attributes and need not be unique; and (b) there should be support in the parsing software to look up references and extract the relevant element with matching `ID`.

It is further recommended in the VOTable 1.2 spec:

While the `ID` attribute has to be unique in a VOTable document, the `name` attribute need not. It is however recommended, as a good practice, to assign unique names within a `TABLE` element. This recommendation means that, between a `TABLE` and its corresponding closing `TABLE` tag, `name` attributes of `FIELD`, `PARAM` and optional `GROUP` elements should be all different.

Since `vo.table` requires a unique identifier for each of its columns, `ID` is used for the column name when present. However, when `ID` is not present, (since it is not required by the specification) `name` is used instead. However, `name` must be cleansed by replacing invalid characters (such as whitespace) with underscores.

Note: This warning does not indicate that the input file is invalid with respect to the VOTable specification, only that the column names in the record array may not match exactly the `name` attributes specified in the file.

References: [1.1](#), [1.2](#)

W04: content-type ‘x’ must be a valid MIME content type The `content-type` attribute must use MIME content-type syntax as defined in [RFC 2046](#).

The current check for validity is somewhat over-permissive.

References: [1.1](#), [1.2](#)

W05: ‘x’ is not a valid URI The attribute must be a valid URI as defined in [RFC 2396](#).

W06: Invalid UCD ‘x’: explanation This warning is emitted when a `ucd` attribute does not match the syntax of a unified content descriptor.

If the VOTable version is 1.2 or later, the UCD will also be checked to ensure it conforms to the controlled vocabulary defined by UCD1+.

References: [1.1](#), [1.2](#)

W07: Invalid astroYear in x: ‘y’ As astro year field is a Besselian or Julian year matching the regular expression:

```
^[JB]?[0-9]+([\.[0-9]*)?$
```

Defined in this XML Schema snippet:

```
<xs:simpleType name="astroYear">
  <xs:restriction base="xs:token">
    <xs:pattern value="[JB]?[0-9]+([\.[0-9]*)?" />
  </xs:restriction>
</xs:simpleType>
```

W08: ‘x’ must be a str or unicode object To avoid local-dependent number parsing differences, `vo.table` may require a string or unicode string where a numeric type may make more sense.

W09: ID attribute not capitalized The VOTable specification uses the attribute name `ID` (with uppercase letters) to specify unique identifiers. Some VOTable-producing tools use the more standard lowercase `id` instead. `vo.table` accepts `id` and emits this warning when not in `pedantic` mode.

References: [1.1](#), [1.2](#)

W10: Unknown tag ‘x’. Ignoring The parser has encountered an element that does not exist in the specification, or appears in an invalid context. Check the file against the VOTable schema (with a tool such as `xmllint`. If the file validates against the schema, and you still receive this warning, this may indicate a bug in `vo.table`.

References: [1.1](#), [1.2](#)

W11: The `gref` attribute on `LINK` is deprecated in VOTable 1.1 Earlier versions of the VOTable specification used a `gref` attribute on the `LINK` element to specify a [GLU reference](#). New files should specify a `glu:` protocol using the `href` attribute.

Since `vo.table` does not currently support GLU references, it likewise does not automatically convert the `gref` attribute to the new form.

References: [1.1](#), [1.2](#)

W12: ‘x’ element must have at least one of ‘ID’ or ‘name’ attributes In order to name the columns of the Numpy record array, each `FIELD` element must have either an `ID` or `name` attribute to derive a name from. Strictly speaking, according to the VOTable schema, the `name` attribute is required. However, if `name` is not present by `ID` is, and *pedantic mode* is off, `vo.table` will continue without a `name` defined.

References: [1.1](#), [1.2](#)

W13: ‘x’ is not a valid VOTable datatype, should be ‘y’ Some VOTable files in the wild use non-standard datatype names. These are mapped to standard ones using the following mapping:

```
string          -> char
unicodeString  -> unicodeChar
int16           -> short
int32           -> int
int64           -> long
float32         -> float
float64         -> double
```

References: [1.1](#), [1.2](#)

W15: x element missing required ‘name’ attribute The `name` attribute is required on every `FIELD` element. However, many VOTable files in the wild omit it and provide only an `ID` instead. In this case, when *pedantic mode* is off, `vo.table` will copy the `name` attribute to a new `ID` attribute.

References: [1.1](#), [1.2](#)

W17: x element contains more than one `DESCRIPTION` element A `DESCRIPTION` element can only appear once within its parent element.

According to the schema, it may only occur once ([1.1](#), [1.2](#))

However, it is a [proposed extension](#) to VOTable 1.2.

W18: `TABLE` specified `nrows=x`, but table contains y rows The number of rows explicitly specified in the `nrows` attribute does not match the actual number of rows (`TR` elements) present in the `TABLE`. This may indicate truncation of the file, or an internal error in the tool that produced it. If *pedantic mode* is off, parsing will proceed, with the loss of some performance.

References: [1.1](#), [1.2](#)

W19: The fields defined in the VOTable do not match those in the embedded FITS file The column fields as defined using `FIELD` elements do not match those in the headers of the embedded FITS file. If *pedantic mode* is off, the embedded FITS file will take precedence.

W20: No version number specified in file. Assuming 1.1 If no version number is explicitly given in the VOTable file, the parser assumes it is written to the VOTable 1.1 specification.

W21: vo.table is designed for VOTable version 1.1, 1.2 and 1.3, but this file is x Unknown issues may arise using `vo.table` with VOTable files from a version other than 1.1, 1.2 or 1.3.

W22: The DEFINITIONS element is deprecated in VOTable 1.1. Ignoring Version 1.0 of the VOTable specification used the `DEFINITIONS` element to define coordinate systems. Version 1.1 now uses `COOSYS` elements throughout the document.

References: [1.1](#), [1.2](#)

W23: Unable to update service information for ‘x’ Raised when the VO service database can not be updated (possibly due to a network outage). This is only a warning, since an older and possible out-of-date VO service database was available locally.

W24: The VO catalog database is for a later version of vo.table The VO catalog database retrieved from the `www` is designed for a newer version of `vo.table`. This may cause problems or limited features performing service queries. Consider upgrading `vo.table` to the latest version.

W25: ‘service’ failed with: ... A VO service query failed due to a network error or malformed arguments. Another alternative service may be attempted. If all services fail, an exception will be raised.

W26: ‘child’ inside ‘parent’ added in VOTable X.X The given element was not supported inside of the given element until the specified VOTable version, however the version declared in the file is for an earlier version. These attributes may not be written out to the file.

W27: COOSYS deprecated in VOTable 1.2 The `COOSYS` element was deprecated in VOTABLE version 1.2 in favor of a reference to the Space-Time Coordinate (STC) data model (see `utype` and the IVOA note [referencing STC in VOTable](#)).

W28: ‘attribute’ on ‘element’ added in VOTable X.X The given attribute was not supported on the given element until the specified VOTable version, however the version declared in the file is for an earlier version. These attributes may not be written out to the file.

W29: Version specified in non-standard form ‘v1.0’ Some VOTable files specify their version number in the form “v1.0”, when the only supported forms in the spec are “1.0”.

References: [1.1](#), [1.2](#)

W30: Invalid literal for float ‘x’. Treating as empty. Some VOTable files write missing floating-point values in non-standard ways, such as “null” and “-”. In non-pedantic mode, any non-standard floating-point literals are treated as missing values.

References: [1.1](#), [1.2](#)

W31: NaN given in an integral field without a specified null value Since NaN's can not be represented in integer fields directly, a null value must be specified in the FIELD descriptor to support reading NaN's from the tabledata.

References: [1.1](#), [1.2](#)

W32: Duplicate ID 'x' renamed to 'x_2' to ensure uniqueness Each field in a table must have a unique ID. If two or more fields have the same ID, some will be renamed to ensure that all IDs are unique.

From the VOTable 1.2 spec:

The `ID` and `ref` attributes are defined as XML types `ID` and `IDREF` respectively. This means that the contents of `ID` is an identifier which must be unique throughout a VOTable document, and that the contents of the `ref` attribute represents a reference to an identifier which must exist in the VOTable document.

References: [1.1](#), [1.2](#)

W33: Column name 'x' renamed to 'x_2' to ensure uniqueness Each field in a table must have a unique name. If two or more fields have the same name, some will be renamed to ensure that all names are unique.

References: [1.1](#), [1.2](#)

W34: 'x' is an invalid token for attribute 'y' The attribute requires the value to be a valid XML token, as defined by [XML 1.0](#).

W35: 'x' attribute required for INFO elements The `name` and `value` attributes are required on all `INFO` elements.

References: [1.1](#), [1.2](#)

W36: null value 'x' does not match field datatype, setting to 0 If the field specifies a `null` value, that value must conform to the given `datatype`.

References: [1.1](#), [1.2](#)

W37: Unsupported data format 'x' The 3 datatypes defined in the VOTable specification and supported by `vo.table` are `TABLEDATA`, `BINARY` and `FITS`.

References: [1.1](#), [1.2](#)

W38: Inline binary data must be base64 encoded, got 'x' The only encoding for local binary data supported by the VOTable specification is `base64`.

W39: Bit values can not be masked Bit values do not support masking. This warning is raised upon setting masked data in a bit column.

References: [1.1](#), [1.2](#)

W40: 'cprojection' datatype repaired This is a terrible hack to support Simple Image Access Protocol results from archive.noao.edu. It creates a field for the coordinate projection type of type "double", which actually contains character data. We have to hack the field to store character data, or we can't read it in. A warning will be raised when this happens.

W41: An XML namespace is specified, but is incorrect. Expected ‘x’, got ‘y’ An XML namespace was specified on the `VOTABLE` element, but the namespace does not match what is expected for a `VOTABLE` file.

The `VOTABLE` namespace is:

```
http://www.ivoa.net/xml/VOTable/vX.X
```

where “X.X” is the version number.

Some files in the wild set the namespace to the location of the `VOTable` schema, which is not correct and will not pass some validating parsers.

W42: No XML namespace specified The root element should specify a namespace.

The `VOTABLE` namespace is:

```
http://www.ivoa.net/xml/VOTable/vX.X
```

where “X.X” is the version number.

W43: element ref=‘x’ which has not already been defined Referenced elements should be defined before referees. From the `VOTable 1.2` spec:

In `VOTable1.2`, it is further recommended to place the `ID` attribute prior to referencing it whenever possible.

W44: VALUES element with ref attribute has content (‘element’) `VALUES` elements that reference another element should not have their own content.

From the `VOTable 1.2` spec:

The `ref` attribute of a `VALUES` element can be used to avoid a repetition of the domain definition, by referring to a previously defined `VALUES` element having the referenced `ID` attribute. When specified, the `ref` attribute defines completely the domain without any other element or attribute, as e.g. `<VALUES ref="RAdomain"/>`

W45: content-role attribute ‘x’ invalid The `content-role` attribute on the `LINK` element must be one of the following:

```
query, hints, doc, location
```

And in `VOTable 1.3`, additionally:

```
type
```

References: [1.1](#), [1.2](#) [1.3](#)

W46: char or unicode value is too long for specified length of x The given char or unicode string is too long for the specified field length.

W47: Missing arraysize indicates length 1 If no `arraysize` is specified on a char field, the default of ‘1’ is implied, but this is rarely what is intended.

W48: Unknown attribute ‘attribute’ on element The attribute is not defined in the specification.

W49: Empty cell illegal for integer fields. Prior to VOTable 1.3, the empty cell was illegal for integer fields.

If a “null” value was specified for the cell, it will be used for the value, otherwise, 0 will be used.

W50: Invalid unit string ‘x’ Invalid unit string as defined in the [Standards for Astronomical Catalogues, Version 2.0](#).

Consider passing an explicit `unit_format` parameter if the units in this file conform to another specification.

W51: Value ‘x’ is out of range for a n-bit integer field The integer value is out of range for the size of the field.

W52: The BINARY2 format was introduced in VOTable 1.3, but this file is declared as version ‘1.2’ The BINARY2 format was introduced in VOTable 1.3. It should not be present in files marked as an earlier version.

W53: VOTABLE element must contain at least one RESOURCE element. The VOTABLE element must contain at least one RESOURCE element.

Exceptions

Note: This is a list of many of the fatal exceptions emitted by `vo.table` when the file does not conform to spec. Other exceptions may be raised due to unforeseen cases or bugs in `vo.table` itself.

E01: Invalid size specifier ‘x’ for a char/unicode field (in field ‘y’) The size specifier for a `char` or `unicode` field must be only a number followed, optionally, by an asterisk. Multi-dimensional size specifiers are not supported for these datatypes.

Strings, which are defined as a set of characters, can be represented in VOTable as a fixed- or variable-length array of characters:

```
<FIELD name="unboundedString" datatype="char" arraysize="*" />
```

A 1D array of strings can be represented as a 2D array of characters, but given the logic above, it is possible to define a variable-length array of fixed-length strings, but not a fixed-length array of variable-length strings.

E02: Incorrect number of elements in array. Expected multiple of x, got y The number of array elements in the data does not match that specified in the FIELD specifier.

E03: ‘x’ does not parse as a complex number Complex numbers should be two values separated by whitespace.

References: [1.1](#), [1.2](#)

E04: Invalid bit value ‘x’ A `bit` array should be a string of ‘0’s and ‘1’s.

References: [1.1](#), [1.2](#)

E05: Invalid boolean value ‘x’ A `boolean` value should be one of the following strings (case insensitive) in the TABLEDATA format:

```
'TRUE', 'FALSE', '1', '0', 'T', 'F', '\0', ' ', '?'
```

and in BINARY format:

```
'T', 'F', '1', '0', '\0', ' ', '?'
```

References: 1.1, 1.2

E06: Unknown datatype ‘x’ on field ‘y’ The supported datatypes are:

```
double, float, bit, boolean, unsignedByte, short, int, long,
floatComplex, doubleComplex, char, unicodeChar
```

The following non-standard aliases are also supported, but in these case *W13* will be raised:

```
string          -> char
unicodeString  -> unicodeChar
int16           -> short
int32           -> int
int64           -> long
float32         -> float
float64         -> double
```

References: 1.1, 1.2

E08: type must be ‘legal’ or ‘actual’, but is ‘x’ The type attribute on the VALUES element must be either legal or actual.

References: 1.1, 1.2

E09: ‘x’ must have a value attribute The MIN, MAX and OPTION elements must always have a value attribute.

References: 1.1, 1.2

E10: ‘datatype’ attribute required on all ‘FIELD’ elements From VOTable 1.1 and later, FIELD and PARAM elements must have a datatype field.

References: 1.1, 1.2

E11: precision ‘x’ is invalid The precision attribute is meant to express the number of significant digits, either as a number of decimal places (e.g. precision="F2" or equivalently precision="2" to express 2 significant figures after the decimal point), or as a number of significant figures (e.g. precision="E5" indicates a relative precision of 10⁻⁵).

It is validated using the following regular expression:

```
[EF]?[1-9][0-9]*
```

References: 1.1, 1.2

E12: width must be a positive integer, got ‘x’ The width attribute is meant to indicate to the application the number of characters to be used for input or output of the quantity.

References: 1.1, 1.2

E13: Invalid arraysize attribute ‘x’ From the VOTable 1.2 spec:

A table cell can contain an array of a given primitive type, with a fixed or variable number of elements; the array may even be multidimensional. For instance, the position of a point in a 3D space can be defined by the following:

```
<FIELD ID="point_3D" datatype="double" arraysize="3"/>
```

and each cell corresponding to that definition must contain exactly 3 numbers. An asterisk (*) may be appended to indicate a variable number of elements in the array, as in:

```
<FIELD ID="values" datatype="int" arraysize="100*"/>
```

where it is specified that each cell corresponding to that definition contains 0 to 100 integer numbers. The number may be omitted to specify an unbounded array (in practice up to $\approx 2 \times 10^9$ elements).

A table cell can also contain a multidimensional array of a given primitive type. This is specified by a sequence of dimensions separated by the x character, with the first dimension changing fastest; as in the case of a simple array, the last dimension may be variable in length. As an example, the following definition declares a table cell which may contain a set of up to 10 images, each of 64×64 bytes:

```
<FIELD ID="thumbs" datatype="unsignedByte" arraysize="64×64×10*"/>
```

References: 1.1, 1.2

E14: value attribute is required for all PARAM elements All PARAM elements must have a value attribute.

References: 1.1, 1.2

E15: ID attribute is required for all COOSYS elements All COOSYS elements must have an ID attribute.

Note that the VOTable 1.1 specification says this attribute is optional, but its corresponding schema indicates it is required.

In VOTable 1.2, the COOSYS element is deprecated.

E16: Invalid system attribute ‘x’ The system attribute on the COOSYS element must be one of the following:

```
'eq_FK4', 'eq_FK5', 'ICRS', 'ecl_FK4', 'ecl_FK5', 'galactic',  
'supergalactic', 'xy', 'barycentric', 'geo_app'
```

References: 1.1

E17: extnum must be a positive integer extnum attribute must be a positive integer.

References: 1.1, 1.2

E18: type must be ‘results’ or ‘meta’, not ‘x’ The type attribute of the RESOURCE element must be one of “results” or “meta”.

References: 1.1, 1.2

E19: File does not appear to be a VOTABLE Raised either when the file doesn’t appear to be XML, or the root element is not VOTABLE.

E20: Data has more columns than are defined in the header (x) The table had only x fields defined, but the data itself has more columns than that.

E21: Data has fewer columns (x) than are defined in the header (y) The table had x fields defined, but the data itself has only y columns.

Exception utilities

`astropy.io.votable.exceptions.warn_or_raise` (*warning_class*, *exception_class=None*,
args=(), *config=None*, *pos=None*, *stack-
level=1*)

Warn or raise an exception, depending on the pedantic setting.

`astropy.io.votable.exceptions.vo_raise` (*exception_class*, *args=()*, *config=None*, *pos=None*)

Raise an exception, with proper position information if available.

`astropy.io.votable.exceptions.vo_reraise` (*exc*, *config=None*, *pos=None*, *additional='u'*)

Raise an exception, with proper position information if available.

Restores the original traceback of the exception, and should only be called within an “except:” block of code.

`astropy.io.votable.exceptions.vo_warn` (*warning_class*, *args=()*, *config=None*, *pos=None*,
stacklevel=1)

Warn, with proper position information if available.

`astropy.io.votable.exceptions.parse_vowarning` (*line*)

Parses the vo warning string back into its parts.

class `astropy.io.votable.exceptions.VOWarning` (*args*, *config=None*, *pos=None*)

Bases: `astropy.utils.exceptions.AstropyWarning`

The base class of all VO warnings and exceptions.

Handles the formatting of the message with a warning or exception code, filename, line and column number.

class `astropy.io.votable.exceptions.VOTableChangeWarning` (*args*, *config=None*,
pos=None)

Bases: `astropy.io.votable.exceptions.VOWarning`, `exceptions.SyntaxWarning`

A change has been made to the input XML file.

class `astropy.io.votable.exceptions.VOTableSpecWarning` (*args*, *config=None*, *pos=None*)

Bases: `astropy.io.votable.exceptions.VOWarning`, `exceptions.SyntaxWarning`

The input XML file violates the spec, but there is an obvious workaround.

class `astropy.io.votable.exceptions.UnimplementedWarning` (*args*, *config=None*,
pos=None)

Bases: `astropy.io.votable.exceptions.VOWarning`, `exceptions.SyntaxWarning`

A feature of the `VOTABLE` spec is not implemented.

class `astropy.io.votable.exceptions.IOWarning` (*args*, *config=None*, *pos=None*)

Bases: `astropy.io.votable.exceptions.VOWarning`, `exceptions.RuntimeWarning`

A network or IO error occurred, but was recovered using the cache.

class `astropy.io.votable.exceptions.VOTableSpecError` (*args*, *config=None*, *pos=None*)

Bases: `astropy.io.votable.exceptions.VOWarning`, `exceptions.ValueError`

The input XML file violates the spec and there is no good workaround.

MISCELLANEOUS INPUT/OUTPUT (ASTROPY.IO.MISC)

The `astropy.io.misc` module contains miscellaneous input/output routines that do not fit elsewhere, and are often used by other Astropy sub-packages. For example, `astropy.io.misc.hdf5` contains functions to read/write `Table` objects from/to HDF5 files, but these should not be imported directly by users. Instead, users can access this functionality via the `Table` class itself (see *Unified file read/write interface*). Routines that are intended to be used directly by users are listed in the `astropy.io.misc` section.

17.1 astropy.io.misc Module

This package contains miscellaneous utility functions for data input/output with astropy.

17.1.1 Functions

<code>fnpickle(object, filename[, usecPickle, ...])</code>	Pickle an object to a specified file.
<code>fnunpickle(filename[, number, usecPickle])</code>	Unpickle pickled objects from a specified file and return the contents.

fnpickle

`astropy.io.misc.fnpickle` (*object, filename, usecPickle=True, protocol=None, append=False*)
Pickle an object to a specified file.

Parameters

object

The python object to pickle.

filename : str or file-like

The filename or file into which the `object` should be pickled. If a file object, it should have been opened in binary mode.

usecPickle : bool

If True (default), the `cPickle` module is to be used in place of `pickle` (`cPickle` is faster). This only applies for python 2.x.

protocol : int or None

Pickle protocol to use - see the `pickle` module for details on these options. If None, the most recent protocol will be used.

append : bool

If True, the object is appended to the end of the file, otherwise the file will be overwritten (if a file object is given instead of a file name, this has no effect).

fnunpickle

`astropy.io.misc.fnunpickle` (*filename*, *number=0*, *usecPickle=True*)

Unpickle pickled objects from a specified file and return the contents.

Parameters

filename : str or file-like

The file name or file from which to unpickle objects. If a file object, it should have been opened in binary mode.

number : int

If 0, a single object will be returned (the first in the file). If >0, this specifies the number of objects to be unpickled, and a list will be returned with exactly that many objects. If <0, all objects in the file will be unpickled and returned as a list.

usecPickle : bool

If True, the `cPickle` module is to be used in place of `pickle` (`cPickle` is faster). This only applies for python 2.x.

Returns

contents : obj or list

If `number` is 0, this is a individual object - the first one unpickled from the file. Otherwise, it is a list of objects unpickled from the file.

Raises

EOFError

If `number` is >0 and there are fewer than `number` objects in the pickled file.

17.2 astropy.io.misc.hdf5 Module

This package contains functions for reading and writing HDF5 tables that are not meant to be used directly, but instead are available as readers/writers in `astropy.table`. See *Unified file read/write interface* for more details.

17.2.1 Functions

<code>read_table_hdf5(input[, path])</code>	Read a Table object from an HDF5 file This requires <code>h5py</code> to be installed.
<code>write_table_hdf5(table, output[, path, ...])</code>	Write a Table object to an HDF5 file This requires <code>h5py</code> to be installed.

read_table_hdf5

`astropy.io.misc.hdf5.read_table_hdf5` (*input*, *path=None*)

Read a Table object from an HDF5 file

This requires `h5py` to be installed. If more than one table is present in the HDF5 file or group, the first table is read in and a warning is displayed.

Parameters

input : str or `File` or `Group` or

`Dataset` If a string, the filename to read the table from. If an `h5py` object, either the file or the group object to read the table from.

path : str

The path from which to read the table inside the HDF5 file. This should be relative to the input file or group.

write_table_hdf5

`astropy.io.misc.hdf5.write_table_hdf5` (*table*, *output*, *path=None*, *compression=False*, *append=False*, *overwrite=False*)

Write a Table object to an HDF5 file

This requires `h5py` to be installed.

Parameters

table : `Table`

Data table that is to be written to file.

output : str or `File` or `Group`

If a string, the filename to write the table to. If an `h5py` object, either the file or the group object to write the table to.

path : str

The path to which to write the table inside the HDF5 file. This should be relative to the input file or group.

compression : bool or str or int

Whether to compress the table inside the HDF5 file. If set to `True`, 'gzip' compression is used. If a string is specified, it should be one of 'gzip', 'szip', or 'lzf'. If an integer is specified (in the range 0-9), 'gzip' compression is used, and the integer denotes the compression level.

append : bool

Whether to append the table to an existing HDF5 file.

overwrite : bool

Whether to overwrite any existing file without warning. If `append=True` and `overwrite=True` then only the dataset will be replaced; the file/group will not be overwritten.

Astronomy computations and utilities

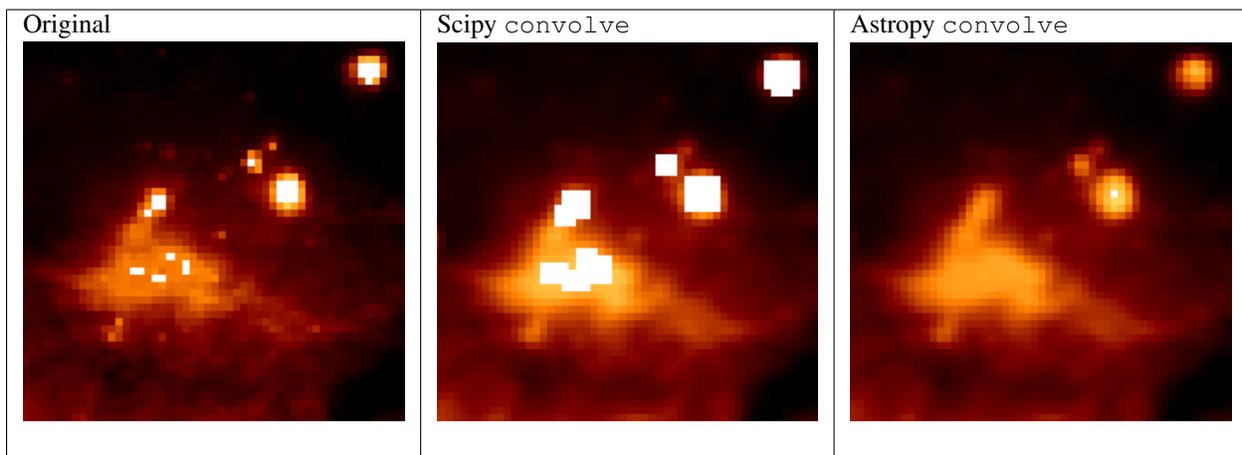
CONVOLUTION AND FILTERING (ASTROPY . CONVOLUTION)

18.1 Introduction

`astropy.convolution` provides convolution functions and kernels that offers improvements compared to the `scipy.ndimage` convolution routines, including:

- Proper treatment of NaN values
- A single function for 1-D, 2-D, and 3-D convolution
- Improved options for the treatment of edges
- Both direct and Fast Fourier Transform (FFT) versions
- Built-in kernels that are commonly used in Astronomy

The following thumbnails show the difference between Scipy's and Astropy's convolve functions on an Astronomical image that contains NaN values. Scipy's function essentially returns NaN for all pixels that are within a kernel of any NaN value, which is often not the desired result.



The following sections describe how to make use of the convolution functions, and how to use built-in convolution kernels:

18.2 Getting started

Two convolution functions are provided. They are imported as:

```
from astropy.convolution import convolve, convolve_fft
```

and are both used as:

```
result = convolve(image, kernel)
result = convolve_fft(image, kernel)
```

`convolve()` is implemented as a direct convolution algorithm, while `convolve_fft()` uses a fast Fourier transform (FFT). Thus, the former is better for small kernels, while the latter is much more efficient for larger kernels.

For example, to convolve a 1-d dataset with a user-specified kernel, you can do:

```
>>> from astropy.convolution import convolve
>>> convolve([1, 4, 5, 6, 5, 7, 8], [0.2, 0.6, 0.2])
array([ 1.4,  3.6,  5. ,  5.6,  5.6,  6.8,  6.2])
```

Notice that the end points are set to zero - by default, points that are too close to the boundary to have a convolved value calculated are set to zero. However, the `convolve()` function allows for a `boundary` argument that can be used to specify alternate behaviors. For example, setting `boundary='extend'` causes values near the edges to be computed, assuming the original data is simply extended using a constant extrapolation beyond the boundary:

```
>>> from astropy.convolution import convolve
>>> convolve([1, 4, 5, 6, 5, 7, 8], [0.2, 0.6, 0.2], boundary='extend')
array([ 1.6,  3.6,  5. ,  5.6,  5.6,  6.8,  7.8])
```

The values at the end are computed assuming that any value below the first point is 1, and any value above the last point is 8. For a more detailed discussion of boundary treatment, see [Using the convolution functions](#).

This module also includes built-in kernels that can be imported as e.g.:

```
>>> from astropy.convolution import Gaussian1DKernel
```

To use a kernel, first create a specific instance of the kernel:

```
>>> gauss = Gaussian1DKernel(stddev=2)
```

`gauss` is not an array, but a kernel object. The underlying array can be retrieved with:

```
>>> gauss.array
array([[ 6.69151129e-05,  4.36341348e-04,  2.21592421e-03,
         8.76415025e-03,  2.69954833e-02,  6.47587978e-02,
         1.20985362e-01,  1.76032663e-01,  1.99471140e-01,
         1.76032663e-01,  1.20985362e-01,  6.47587978e-02,
         2.69954833e-02,  8.76415025e-03,  2.21592421e-03,
         4.36341348e-04,  6.69151129e-05])
```

The kernel can then be used directly when calling `convolve()`:

```
import numpy as np
import matplotlib.pyplot as plt

from astropy.convolution import Gaussian1DKernel, convolve

# Generate fake data
x = np.arange(1000).astype(float)
y = np.sin(x / 100.) + np.random.normal(0., 1., x.shape)

# Create kernel
g = Gaussian1DKernel(stddev=50)

# Convolve data
```

```
z = convolve(y, g, boundary='extend')

# Plot data before and after convolution
plt.plot(x, y, 'k.')
plt.plot(x, z, 'r-', lw=3)
plt.show()
```

18.3 Using `astropy.convolution`

18.3.1 Using the convolution functions

Overview

Two convolution functions are provided. They are imported as:

```
>>> from astropy.convolution import convolve, convolve_fft
```

and are both used as:

```
>>> result = convolve(image, kernel)
>>> result = convolve_fft(image, kernel)
```

`convolve()` is implemented as a direct convolution algorithm, while `convolve_fft()` uses a fast Fourier transform (FFT). Thus, the former is better for small kernels, while the latter is much more efficient for larger kernels.

The input images and kernels should be lists or Numpy arrays with either both 1, 2, or 3 dimensions (and the number of dimensions should be the same for the image and kernel). The result is a Numpy array with the same dimensions as the input image. The convolution is always done as floating point.

The `convolve()` function takes an optional `boundary=` argument describing how to perform the convolution at the edge of the array. The values for `boundary` can be:

- `None`: set the result values to zero where the kernel extends beyond the edge of the array (default)
- `'fill'`: set values outside the array boundary to a constant. If this option is specified, the constant should be specified using the `fill_value=` argument, which defaults to zero.
- `'wrap'`: assume that the boundaries are periodic
- `'extend'`: set values outside the array to the nearest array value

By default, the kernel is not normalized. To normalize it prior to convolution, use:

```
>>> result = convolve(image, kernel, normalize_kernel=True)
```

Examples

Smooth a 1D array with a custom kernel and no boundary treatment:

```
>>> import numpy as np
>>> convolve([1, 4, 5, 6, 5, 7, 8], [0.2, 0.6, 0.2])
array([ 1.4,  3.6,  5. ,  5.6,  5.6,  6.8,  6.2])
```

As above, but using the `'extend'` algorithm for boundaries:

```
>>> convolve([1, 4, 5, 6, 5, 7, 8], [0.2, 0.6, 0.2], boundary='extend')
array([ 1.6,  3.6,  5. ,  5.6,  5.6,  6.8,  7.8])
```

If a NaN value is present in the original array, it will be interpolated using the kernel:

```
>>> import numpy as np
>>> convolve([1, 4, 5, 6, np.nan, 7, 8], [0.2, 0.6, 0.2], boundary='extend')
array([ 1.6,  3.6,  5. ,  5.9,  6.5,  7.1,  7.8])
```

Kernels and arrays can be specified either as lists or as Numpy arrays. The following examples show how to construct a 1-d array as a list:

```
>>> kernel = [0, 1, 0]
>>> result = convolve(spectrum, kernel)
```

a 2-d array as a list:

```
>>> kernel = [[0, 1, 0],
...          [1, 2, 1],
...          [0, 1, 0]]
>>> result = convolve(image, kernel)
```

and a 3-d array as a list:

```
>>> kernel = [[[0, 0, 0], [0, 2, 0], [0, 0, 0]],
...          [[0, 1, 0], [2, 3, 2], [0, 1, 0]],
...          [[0, 0, 0], [0, 2, 0], [0, 0, 0]]]
>>> result = convolve(cube, kernel)
```

Kernels

The above examples uses custom kernels, but `astropy.convolution` also includes a number of built-in kernels, which are described in [Convolution Kernels](#).

18.3.2 Convolution Kernels

Introduction and Concept

The convolution module provides several built-in kernels to cover the most common applications in astronomy. It is also possible to define custom kernels from arrays or combine existing kernels to match specific applications.

Every filter kernel is characterized by its response function. For time series we speak of an “impulse response function” or for images we call it “point spread function”. This response function is given for every kernel by a `FittableModel`, which is evaluated on a grid with `discretize_model()` to obtain a kernel array, which can be used for discrete convolution with the binned data.

Examples

1D Kernels

One application of filtering is to smooth noisy data. In this case we consider a noisy Lorentz curve:

```
>>> import numpy as np
>>> from astropy.modeling.models import Lorentz1D
>>> from astropy.convolution import convolve, Gaussian1DKernel, Box1DKernel
>>> lorentz = Lorentz1D(1, 0, 1)
>>> x = np.linspace(-5, 5, 100)
>>> data_1D = lorentz(x) + 0.1 * (np.random.rand(100) - 0.5)
```

Smoothing the noisy data with a `Gaussian1DKernel` with a standard deviation of 2 pixels:

```
>>> gauss_kernel = Gaussian1DKernel(2)
>>> smoothed_data_gauss = convolve(data_1D, gauss_kernel)
```

Smoothing the same data with a `Box1DKernel` of width 5 pixels:

```
>>> box_kernel = Box1DKernel(5)
>>> smoothed_data_box = convolve(data_1D, box_kernel)
```

The following plot illustrates the results:

Beside the astropy convolution functions `convolve` and `convolve_fft`, it is also possible to use the kernels with Numpy or Scipy convolution by passing the `array` attribute. This will be faster in most cases than the astropy convolution, but will not work properly if NaN values are present in the data.

```
>>> smoothed = np.convolve(data_1D, box_kernel.array)
```

2D Kernels

As all 2D kernels are symmetric it is sufficient to specify the width in one direction. Therefore the use of 2D kernels is basically the same as for 1D kernels. We consider a small Gaussian shaped source of amplitude one in the middle of the image and add 10% noise:

```
>>> import numpy as np
>>> from astropy.convolution import convolve, Gaussian2DKernel, Tophat2DKernel
>>> from astropy.modeling.models import Gaussian2D
>>> gauss = Gaussian2D(1, 0, 0, 3, 3)
>>> # Fake image data including noise
>>> x = np.arange(-100, 101)
>>> y = np.arange(-100, 101)
>>> x, y = np.meshgrid(x, y)
>>> data_2D = gauss(x, y) + 0.1 * (np.random.rand(201, 201) - 0.5)
```

Smoothing the noisy data with a `Gaussian2DKernel` with a standard deviation of 2 pixels:

```
>>> gauss_kernel = Gaussian2DKernel(2)
>>> smoothed_data_gauss = convolve(data_2D, gauss_kernel)
```

Smoothing the noisy data with a `Tophat2DKernel` of width 5 pixels:

```
>>> tophat_kernel = Tophat2DKernel(5)
>>> smoothed_data_tophat = convolve(data_2D, tophat_kernel)
```

This is what the original image looks like:

The following plot illustrates the differences between several 2D kernels applied to the simulated data. Note that it has a slightly different color scale compared to the original image.

The Gaussian kernel has better smoothing properties compared to the Box and the Tophat. The Box filter is not isotropic and can produce artifact (the source appears rectangular). The Mexican-Hat filter removes noise and slowly varying structures (i.e. background), but produces a negative ring around the source. The best choice for the filter strongly depends on the application.

Available Kernels

<code>AiryDisk2DKernel(radius, **kwargs)</code>	2D Airy disk kernel.
---	----------------------

Continued on next page

Table 18.1 – continued from previous page

<code>Box1DKernel(width, **kwargs)</code>	1D Box filter kernel.
<code>Box2DKernel(width, **kwargs)</code>	2D Box filter kernel.
<code>CustomKernel(array)</code>	Create filter kernel from list or array.
<code>Gaussian1DKernel(stddev, **kwargs)</code>	1D Gaussian filter kernel.
<code>Gaussian2DKernel(stddev, **kwargs)</code>	2D Gaussian filter kernel.
<code>MexicanHat1DKernel(width, **kwargs)</code>	1D Mexican hat filter kernel.
<code>MexicanHat2DKernel(width, **kwargs)</code>	2D Mexican hat filter kernel.
<code>Model1DKernel(model, **kwargs)</code>	Create kernel from 1D model.
<code>Model2DKernel(model, **kwargs)</code>	Create kernel from 2D model.
<code>Ring2DKernel(radius_in, width, **kwargs)</code>	2D Ring filter kernel.
<code>Tophat2DKernel(radius, **kwargs)</code>	2D Tophat filter kernel.
<code>Trapezoid1DKernel(width[, slope])</code>	1D trapezoid kernel.
<code>TrapezoidDisk2DKernel(radius[, slope])</code>	2D trapezoid kernel.

Kernel Arithmetics

Addition and Subtraction

As convolution is a linear operation, kernels can be added or subtracted from each other. They can also be multiplied with some number. One basic example would be the definition of a Difference of Gaussian filter:

```
>>> from astropy.convolution import Gaussian1DKernel
>>> gauss_1 = Gaussian1DKernel(10)
>>> gauss_2 = Gaussian1DKernel(16)
>>> DoG = gauss_2 - gauss_1
```

Another application is to convolve faked data with an instrument response function model. E.g. if the response function can be described by the weighted sum of two Gaussians:

```
>>> gauss_1 = Gaussian1DKernel(10)
>>> gauss_2 = Gaussian1DKernel(16)
>>> SoG = 4 * gauss_1 + gauss_2
```

Most times it will be necessary to normalize the resulting kernel by calling explicitly:

```
>>> SoG.normalize()
```

Convolution

Furthermore two kernels can be convolved with each other, which is useful when data is filtered with two different kinds of kernels or to create a new, special kernel:

```
>>> from astropy.convolution import Gaussian1DKernel, convolve
>>> gauss_1 = Gaussian1DKernel(10)
>>> gauss_2 = Gaussian1DKernel(16)
>>> broad_gaussian = convolve(gauss_2, gauss_1)
```

Or in case of multistage smoothing:

```
>>> import numpy as np
>>> from astropy.modeling.models import Lorentz1D
>>> from astropy.convolution import convolve, Gaussian1DKernel, Box1DKernel
>>> lorentz = Lorentz1D(1, 0, 1)
>>> x = np.linspace(-5, 5, 100)
>>> data_1D = lorentz(x) + 0.1 * (np.random.rand(100) - 0.5)
```

```
>>> gauss = Gaussian1DKernel(3)
>>> box = Box1DKernel(5)
>>> smoothed_gauss = convolve(data_1D, gauss)
>>> smoothed_gauss_box = convolve(smoothed_gauss, box)
```

You would rather do the following:

```
>>> gauss = Gaussian1DKernel(3)
>>> box = Box1DKernel(5)
>>> smoothed_gauss_box = convolve(data_1D, convolve(box, gauss))
```

Which, in most cases, will also be faster than the first method, because only one convolution with the, most times, larger data array will be necessary.

Discretization

To obtain the kernel array for discrete convolution, the kernels response function is evaluated on a grid with `discretize_model()`. For the discretization step the following modes are available:

- Mode 'center' (default) evaluates the response function on the grid by taking the value at the center of the bin.

```
>>> from astropy.convolution import Gaussian1DKernel
>>> gauss_center = Gaussian1DKernel(3, mode='center')
```

- Mode 'linear_interp' takes the values at the corners of the bin and linearly interpolates the value at the center:

```
>>> gauss_interp = Gaussian1DKernel(3, mode='linear_interp')
```

- Mode 'oversample' evaluates the response function by taking the mean on an oversampled grid. The oversample factor can be specified with the `factor` argument. If the oversample factor is too large, the evaluation becomes slow.

```
>>> gauss_oversample = Gaussian1DKernel(3, mode='oversample', factor=10)
```

- Mode 'integrate' integrates the function over the pixel using `scipy.integrate.quad` and `scipy.integrate.dblquad`. This mode is very slow and only recommended when highest accuracy is required.

```
>>> gauss_integrate = Gaussian1DKernel(3, mode='integrate')
```

Especially in the range where the kernel width is in order of only a few pixels it can be advantageous to use the mode `oversample` or `integrate` to conserve the integral on a subpixel scale.

Normalization

The kernel models are normalized per default, i.e. $\int_{-\infty}^{\infty} f(x)dx = 1$. But because of the limited kernel array size the normalization for kernels with an infinite response can differ from one. The value of this deviation is stored in the kernel's `truncation` attribute.

The normalization can also differ from one, especially for small kernels, due to the discretization step. This can be partly controlled by the `mode` argument, when initializing the kernel (See also `discretize_model()`). Setting the mode to 'oversample' allows to conserve the normalization even on the subpixel scale.

The kernel arrays can be renormalized explicitly by calling either the `normalize()` method or by setting the `normalize_kernel` argument in the `convolve()` and `convolve_fft()` functions. The latter method leaves the kernel itself unchanged but works with an internal normalized version of the kernel.

Note that for `MexicanHat1DKernel` and `MexicanHat2DKernel` there is $\int_{-\infty}^{\infty} f(x)dx = 0$. To define a proper normalization both filters are derived from a normalized Gaussian function.

18.4 Reference/API

18.4.1 `astropy.convolution` Module

Functions

<code>convolve(array, kernel[, boundary, ...])</code>	Convolve an array with a kernel.
<code>convolve_fft(array, kernel[, boundary, ...])</code>	Convolve an ndarray with an nd-kernel.
<code>discretize_model(model, x_range[, y_range, ...])</code>	Function to evaluate analytical models on a grid.
<code>kernel_arithmetics(kernel, value, operation)</code>	Add, subtract or multiply two kernels.

`convolve`

`astropy.convolution.convolve(array, kernel, boundary='fill', fill_value=0.0, normalize_kernel=False)`

Convolve an array with a kernel.

This routine differs from `scipy.ndimage.filters.convolve` because it includes a special treatment for NaN values. Rather than including NaN's in the convolution calculation, which causes large NaN holes in the convolved image, NaN values are replaced with interpolated values using the kernel as an interpolation function.

Parameters

array : `numpy.ndarray`

The array to convolve. This should be a 1, 2, or 3-dimensional array or a list or a set of nested lists representing a 1, 2, or 3-dimensional array.

kernel : `numpy.ndarray` or `Kernel`

The convolution kernel. The number of dimensions should match those for the array, and the dimensions should be odd in all directions.

boundary : str, optional

A flag indicating how to handle boundaries:

- None**

Set the result values to zero where the kernel extends beyond the edge of the array (default).

- 'fill'**

Set values outside the array boundary to `fill_value`.

- 'wrap'**

Periodic boundary that wrap to the other side of array.

- 'extend'**

Set values outside the array to the nearest array value.

fill_value : float, optional

The value to use outside the array when using `boundary='fill'`

normalize_kernel : bool, optional

Whether to normalize the kernel prior to convolving

Returns

result : `numpy.ndarray`

An array with the same dimensions and as the input array, convolved with kernel. The data type depends on the input array type. If array is a floating point type, then the return array keeps the same data type, otherwise the type is `numpy.float`.

Notes

Masked arrays are not supported at this time. The convolution is always done at `numpy.float` precision.

convolve_fft

```
astropy.convolution.convolve_fft(array, kernel, boundary='fill', fill_value=0, crop=True,
                                return_fft=False, fft_pad=True, psf_pad=False, interpolate_nan=False,
                                quiet=False, ignore_edge_zeros=False, min_wt=0.0, normalize_kernel=False,
                                allow_huge=False, fftn=<function fftn at 0x24609b0>, ifftn=<function ifftn at
                                0x2460a28>, complex_dtype=<type 'complex'>)
```

Convolve an ndarray with an nd-kernel. Returns a convolved image with shape = array.shape. Assumes kernel is centered.

`convolve_fft` differs from `scipy.signal.fftconvolve` in a few ways:

- It can treat NaN values as zeros or interpolate over them.
- `inf` values are treated as NaN
- (optionally) It pads to the nearest 2^n size to improve FFT speed.
- Its only valid mode is 'same' (i.e., the same shape array is returned)
- It lets you use your own fft, e.g., `pyFFTW` or `pyFFTW3`, which can lead to performance improvements, depending on your system configuration. `pyFFTW3` is threaded, and therefore may yield significant performance benefits on multi-core machines at the cost of greater memory requirements. Specify the `fftn` and `ifftn` keywords to override the default, which is `numpy.fft.fft` and `numpy.fft.ifft`.

Parameters

array : `numpy.ndarray`

Array to be convolved with kernel

kernel : `numpy.ndarray`

Will be normalized if `normalize_kernel` is set. Assumed to be centered (i.e., shifts may result if your kernel is asymmetric)

boundary : {'fill', 'wrap'}, optional

A flag indicating how to handle boundaries:

- 'fill': set values outside the array boundary to `fill_value` (default)
- 'wrap': periodic boundary

interpolate_nan : bool, optional

The convolution will be re-weighted assuming NaN values are meant to be ignored, not treated as zero. If this is off, all NaN values will be treated as zero.

ignore_edge_zeros : bool, optional

Ignore the zero-pad-created zeros. This will effectively decrease the kernel area on the edges but will not re-normalize the kernel. This parameter may result in ‘edge-brightening’ effects if you’re using a normalized kernel

min_wt : float, optional

If ignoring NaN / zeros, force all grid points with a weight less than this value to NaN (the weight of a grid point with *no* ignored neighbors is 1.0). If `min_wt` is zero, then all zero-weight points will be set to zero instead of NaN (which they would be otherwise, because $1/0 = \text{nan}$). See the examples below

normalize_kernel : function or boolean, optional

If specified, this is the function to divide kernel by to normalize it. e.g., `normalize_kernel=np.sum` means that kernel will be modified to be: `kernel = kernel / np.sum(kernel)`. If True, defaults to `normalize_kernel = np.sum`.

Returns

default : ndarray

array convolved with `kernel`. If `return_fft` is set, returns `fft(array) * fft(kernel)`. If `crop` is not set, returns the image, but with the fft-padded size instead of the input size

Other Parameters

fft_pad : bool, optional

Default on. Zero-pad image to the nearest 2^n

psf_pad : bool, optional

Default off. Zero-pad image to be at least the sum of the image sizes (in order to avoid edge-wrapping when smoothing)

crop : bool, optional

Default on. Return an image of the size of the largest input image. If the images are asymmetric in opposite directions, will return the largest image in both directions. For example, if an input image has shape [100,3] but a kernel with shape [6,6] is used, the output will be [100,6].

return_fft : bool, optional

Return the `fft(image)*fft(kernel)` instead of the convolution (which is `ifft(fft(image)*fft(kernel))`). Useful for making PSDs.

fftn, ifftn : functions, optional

The fft and inverse fft functions. Can be overridden to use your own ffts, e.g. an `fftw3` wrapper or `scipy`’s `fftn`, e.g. `fftn=scipy.fftpack.fftn`

complex_dtype : np.complex, optional

Which complex dtype to use. `numpy` has a range of options, from 64 to 256.

quiet : bool, optional

Silence warning message about NaN interpolation

allow_huge : bool, optional

Allow huge arrays in the FFT? If False, will raise an exception if the array or kernel size is >1 GB

Raises

ValueError:

If the array is bigger than 1 GB after padding, will raise this exception unless `allow_huge` is True

See also:

`convolve`

Convolve is a non-fft version of this code. It is more memory efficient and for small kernels can be faster.

Examples

```
>>> convolve_fft([1, 0, 3], [1, 1, 1])
array([ 1.,  4.,  3.])

>>> convolve_fft([1, np.nan, 3], [1, 1, 1])
array([ 1.,  4.,  3.])

>>> convolve_fft([1, 0, 3], [0, 1, 0])
array([ 1.,  0.,  3.])

>>> convolve_fft([1, 2, 3], [1])
array([ 1.,  2.,  3.])

>>> convolve_fft([1, np.nan, 3], [0, 1, 0], interpolate_nan=True)
...
array([ 1.,  0.,  3.])

>>> convolve_fft([1, np.nan, 3], [0, 1, 0], interpolate_nan=True,
...              min_wt=1e-8)
array([ 1.,  nan,  3.])

>>> convolve_fft([1, np.nan, 3], [1, 1, 1], interpolate_nan=True)
array([ 1.,  4.,  3.])

>>> convolve_fft([1, np.nan, 3], [1, 1, 1], interpolate_nan=True,
...              normalize_kernel=True, ignore_edge_zeros=True)
array([ 1.,  2.,  3.])

>>> import scipy.fftpack # optional - requires scipy
>>> convolve_fft([1, np.nan, 3], [1, 1, 1], interpolate_nan=True,
...              normalize_kernel=True, ignore_edge_zeros=True,
...              fftn=scipy.fftpack.fft, ifftn=scipy.fftpack.ifft)
array([ 1.,  2.,  3.])
```

`discretize_model`

`astropy.convolution.discretize_model` (*model*, *x_range*, *y_range=None*, *mode='center'*, *factor=10*)

Function to evaluate analytical models on a grid.

Parameters**model** : `FittableModel`

Model to be evaluated.

x_range : tuple

x range in which the model is evaluated.

y_range : tuple, optional

y range in which the model is evaluated. Necessary only for 2D models.

mode : str, optional**One of the following modes:**• **'center'** (default)

Discretize model by taking the value at the center of the bin.

• **'linear_interp'**

Discretize model by linearly interpolating between the values at the corners of the bin. For 2D models interpolation is bilinear.

• **'oversample'**

Discretize model by taking the average on an oversampled grid.

• **'integrate'**Discretize model by integrating the model over the bin using `scipy.integrate.quad`. Very slow.**factor** : float or int

Factor of oversampling. Default = 10.

Returns**array** : `numpy.array`

Model value array

Notes

The `oversample` mode allows to conserve the integral on a subpixel scale. Here is the example of a normalized Gaussian1D:

```
import matplotlib.pyplot as plt
import numpy as np
from astropy.modeling.models import Gaussian1D
from astropy.convolution.utils import discretize_model
gauss_1D = Gaussian1D(1 / (0.5 * np.sqrt(2 * np.pi)), 0, 0.5)
y_center = discretize_model(gauss_1D, (-2, 3), mode='center')
y_corner = discretize_model(gauss_1D, (-2, 3), mode='linear_interp')
y_oversample = discretize_model(gauss_1D, (-2, 3), mode='oversample')
plt.plot(y_center, label='center sum = {0:3f}'.format(y_center.sum()))
plt.plot(y_corner, label='linear_interp sum = {0:3f}'.format(y_corner.sum()))
plt.plot(y_oversample, label='oversample sum = {0:3f}'.format(y_oversample.sum()))
plt.xlabel('pixels')
plt.ylabel('value')
plt.legend()
plt.show()
```

kernel_arithmetics

`astropy.convolution.kernel_arithmetics` (*kernel, value, operation*)

Add, subtract or multiply two kernels.

Parameters

kernel : `astropy.convolution.Kernel`

Kernel instance

value : kernel, float or int

Value to operate with

operation : {'add', 'sub', 'mul'}

One of the following operations:

•**add**

Add two kernels

•**sub**

Subtract two kernels

•**mul**

Multiply kernel with number or convolve two kernels.

Classes

<code>AiryDisk2DKernel(radius, **kwargs)</code>	2D Airy disk kernel.
<code>Box1DKernel(width, **kwargs)</code>	1D Box filter kernel.
<code>Box2DKernel(width, **kwargs)</code>	2D Box filter kernel.
<code>CustomKernel(array)</code>	Create filter kernel from list or array.
<code>Gaussian1DKernel(stddev, **kwargs)</code>	1D Gaussian filter kernel.
<code>Gaussian2DKernel(stddev, **kwargs)</code>	2D Gaussian filter kernel.
<code>Kernel(array)</code>	Convolution kernel base class.
<code>Kernel1D([model, x_size, array])</code>	Base class for 1D filter kernels.
<code>Kernel2D([model, x_size, y_size, array])</code>	Base class for 2D filter kernels.
<code>MexicanHat1DKernel(width, **kwargs)</code>	1D Mexican hat filter kernel.
<code>MexicanHat2DKernel(width, **kwargs)</code>	2D Mexican hat filter kernel.
<code>Model1DKernel(model, **kwargs)</code>	Create kernel from 1D model.
<code>Model2DKernel(model, **kwargs)</code>	Create kernel from 2D model.
<code>Ring2DKernel(radius_in, width, **kwargs)</code>	2D Ring filter kernel.
<code>Tophat2DKernel(radius, **kwargs)</code>	2D Tophat filter kernel.
<code>Trapezoid1DKernel(width[, slope])</code>	1D trapezoid kernel.
<code>TrapezoidDisk2DKernel(radius[, slope])</code>	2D trapezoid kernel.

AiryDisk2DKernel

class `astropy.convolution.AiryDisk2DKernel` (*radius, **kwargs*)

Bases: `astropy.convolution.Kernel2D`

2D Airy disk kernel.

This kernel models the diffraction pattern of a circular aperture. This kernel is normalized to a peak value of 1.

Parameters**radius** : float

The radius of the Airy disk kernel (radius of the first zero).

x_size : odd int, optional

Size in x direction of the kernel array. Default = 8 * radius.

y_size : odd int, optional

Size in y direction of the kernel array. Default = 8 * radius.

mode : str, optional**One of the following discretization modes:**•**‘center’ (default)**

Discretize model by taking the value at the center of the bin.

•**‘linear_interp’**

Discretize model by performing a bilinear interpolation between the values at the corners of the bin.

•**‘oversample’**

Discretize model by taking the average on an oversampled grid.

•**‘integrate’**

Discretize model by integrating the model over the bin.

factor : number, optional

Factor of oversampling. Default factor = 10.

See also:

`Box2DKernel`, `Tophat2DKernel`, `MexicanHat2DKernel`, `Ring2DKernel`,
`TrapezoidDisk2DKernel`, `AiryDisk2DKernel`

Examples

Kernel response:

```
import matplotlib.pyplot as plt
from astropy.convolution import AiryDisk2DKernel
airydisk_2d_kernel = AiryDisk2DKernel(10)
plt.imshow(airydisk_2d_kernel, interpolation='none', origin='lower')
plt.xlabel('x [pixels]')
plt.ylabel('y [pixels]')
plt.colorbar()
plt.show()
```

Box1DKernel**class** `astropy.convolution.Box1DKernel` (*width*, ***kwargs*)Bases: `astropy.convolution.Kernel1D`

1D Box filter kernel.

The Box filter or running mean is a smoothing filter. It is not isotropic and can produce artifacts, when applied repeatedly to the same data.

By default the Box kernel uses the `linear_interp` discretization mode, which allows non-shifting, even-sized kernels. This is achieved by weighting the edge pixels with 1/2. E.g a Box kernel with an effective smoothing of 4 pixel would have the following array: [0.5, 1, 1, 1, 0.5].

Parameters

width : number

Width of the filter kernel.

mode : str, optional

One of the following discretization modes:

- **‘center’**

Discretize model by taking the value at the center of the bin.

- **‘linear_interp’ (default)**

Discretize model by linearly interpolating between the values at the corners of the bin.

- **‘oversample’**

Discretize model by taking the average on an oversampled grid.

- **‘integrate’**

Discretize model by integrating the model over the bin.

factor : number, optional

Factor of oversampling. Default factor = 10.

See also:

[Gaussian1DKernel](#), [Trapezoid1DKernel](#), [MexicanHat1DKernel](#)

Examples

Kernel response function:

```
import matplotlib.pyplot as plt
from astropy.convolution import Box1DKernel
box_1D_kernel = Box1DKernel(9)
plt.plot(box_1D_kernel, drawstyle='steps')
plt.xlim(-1, 9)
plt.xlabel('x [pixels]')
plt.ylabel('value')
plt.show()
```

Box2DKernel

class `astropy.convolution.Box2DKernel` (*width*, ***kwargs*)

Bases: `astropy.convolution.Kernel2D`

2D Box filter kernel.

The Box filter or running mean is a smoothing filter. It is not isotropic and can produce artifact, when applied repeatedly to the same data.

By default the Box kernel uses the `linear_interp` discretization mode, which allows non-shifting, even-sized kernels. This is achieved by weighting the edge pixels with 1/2.

Parameters

width : number

Width of the filter kernel.

mode : str, optional

One of the following discretization modes:

•‘center’

Discretize model by taking the value at the center of the bin.

•‘linear_interp’ (default)

Discretize model by performing a bilinear interpolation between the values at the corners of the bin.

•‘oversample’

Discretize model by taking the average on an oversampled grid.

•‘integrate’

Discretize model by integrating the model over the bin.

factor : number, optional

Factor of oversampling. Default factor = 10.

See also:

`Box2DKernel`, `Tophat2DKernel`, `MexicanHat2DKernel`, `Ring2DKernel`,
`TrapezoidDisk2DKernel`, `AiryDisk2DKernel`

Examples

Kernel response:

```
import matplotlib.pyplot as plt
from astropy.convolution import Box2DKernel
box_2d_kernel = Box2DKernel(9)
plt.imshow(box_2d_kernel, interpolation='none', origin='lower',
           vmin=0.0, vmax=0.015)
plt.xlim(-1, 9)
plt.ylim(-1, 9)
plt.xlabel('x [pixels]')
plt.ylabel('y [pixels]')
plt.colorbar()
plt.show()
```

CustomKernel

class `astropy.convolution.CustomKernel` (*array*)
Bases: `astropy.convolution.Kernel`

Create filter kernel from list or array.

Parameters

array : list or array

Filter kernel array. Size must be odd.

Raises

TypeError

If array is not a list or array.

KernelSizeError

If array size is even.

See also:

`Model2DKernel`, `Model1DKernel`

Examples

Define one dimensional array:

```
>>> from astropy.convolution.kernels import CustomKernel
>>> import numpy as np
>>> array = np.array([1, 2, 3, 2, 1])
>>> kernel = CustomKernel(array)
>>> kernel.dimension
1
```

Define two dimensional array:

```
>>> array = np.array([[1, 1, 1], [1, 2, 1], [1, 1, 1]])
>>> kernel = CustomKernel(array)
>>> kernel.dimension
2
```

Attributes Summary

<code>array</code>	Filter kernel array.
--------------------	----------------------

Attributes Documentation

array

Filter kernel array.

Gaussian1DKernel

class `astropy.convolution.Gaussian1DKernel` (*stddev*, ***kwargs*)

Bases: `astropy.convolution.Kernel1D`

1D Gaussian filter kernel.

The Gaussian filter is a filter with great smoothing properties. It is isotropic and does not produce artifacts.

Parameters

stddev : number

Standard deviation of the Gaussian kernel.

x_size : odd int, optional

Size of the kernel array. Default = $8 * \text{stddev}$

mode : str, optional

One of the following discretization modes:

•**‘center’ (default)**

Discretize model by taking the value at the center of the bin.

•**‘linear_interp’**

Discretize model by linearly interpolating between the values at the corners of the bin.

•**‘oversample’**

Discretize model by taking the average on an oversampled grid.

•**‘integrate’**

Discretize model by integrating the model over the bin. Very slow.

factor : number, optional

Factor of oversampling. Default factor = 10. If the factor is too large, evaluation can be very slow.

See also:

[Box1DKernel](#), [Trapezoid1DKernel](#), [MexicanHat1DKernel](#)

Examples

Kernel response:

```
import matplotlib.pyplot as plt
from astropy.convolution import Gaussian1DKernel
gauss_1d_kernel = Gaussian1DKernel(10)
plt.plot(gauss_1d_kernel, drawstyle='steps')
plt.xlabel('x [pixels]')
plt.ylabel('value')
plt.show()
```

Gaussian2DKernel

class `astropy.convolution.Gaussian2DKernel` (*stddev*, ***kwargs*)

Bases: `astropy.convolution.Kernel2D`

2D Gaussian filter kernel.

The Gaussian filter is a filter with great smoothing properties. It is isotropic and does not produce artifacts.

Parameters

stddev : number

Standard deviation of the Gaussian kernel.

x_size : odd int, optional

Size in x direction of the kernel array. Default = $8 * \text{stddev}$.

y_size : odd int, optional

Size in y direction of the kernel array. Default = $8 * \text{stddev}$.

mode : str, optional

One of the following discretization modes:

- **‘center’ (default)**
Discretize model by taking the value at the center of the bin.
- **‘linear_interp’**
Discretize model by performing a bilinear interpolation between the values at the corners of the bin.
- **‘oversample’**
Discretize model by taking the average on an oversampled grid.
- **‘integrate’**
Discretize model by integrating the model over the bin.

factor : number, optional

Factor of oversampling. Default factor = 10.

See also:

`Box2DKernel`, `Tophat2DKernel`, `MexicanHat2DKernel`, `Ring2DKernel`,
`TrapezoidDisk2DKernel`, `AiryDisk2DKernel`

Examples

Kernel response:

```
import matplotlib.pyplot as plt
from astropy.convolution import Gaussian2DKernel
gaussian_2d_kernel = Gaussian2DKernel(10)
plt.imshow(gaussian_2d_kernel, interpolation='none', origin='lower')
plt.xlabel('x [pixels]')
plt.ylabel('y [pixels]')
plt.colorbar()
plt.show()
```

Kernel

class `astropy.convolution.Kernel` (*array*)

Bases: `object`

Convolution kernel base class.

Parameters

array : `ndarray`

Kernel array.

Attributes Summary

<code>array</code>	Filter kernel array.
<code>center</code>	Index of the kernel center.
Continued on next page	

Table 18.5 – continued from previous page

<code>dimension</code>	Kernel dimension.
<code>is_bool</code>	Indicates if kernel is bool.
<code>model</code>	Kernel response model.
<code>normalization</code>	Kernel normalization factor
<code>separable</code>	Indicates if the filter kernel is separable.
<code>shape</code>	Shape of the kernel array.
<code>truncation</code>	Deviation from the normalization to one.

Methods Summary

<code>normalize([mode])</code>	Force normalization of filter kernel.
--------------------------------	---------------------------------------

Attributes Documentation

array

Filter kernel array.

center

Index of the kernel center.

dimension

Kernel dimension.

is_bool

Indicates if kernel is bool.

If the kernel is bool the multiplication in the convolution could be omitted, to increase the performance.

model

Kernel response model.

normalization

Kernel normalization factor

separable

Indicates if the filter kernel is separable.

A 2D filter is separable, when its filter array can be written as the outer product of two 1D arrays.

If a filter kernel is separable, higher dimension convolutions will be performed by applying the 1D filter array consecutively on every dimension. This is significantly faster, than using a filter array with the same dimension.

shape

Shape of the kernel array.

truncation

Deviation from the normalization to one.

Methods Documentation

normalize (*mode*=*u'integral'*)

Force normalization of filter kernel.

Parameters

mode : { 'integral', 'peak' }

One of the following modes:

- **‘integral’ (default)**
Kernel normalized such that its integral = 1.
- **‘peak’**
Kernel normalized such that its peak = 1.

Kernel1D

class `astropy.convolution.Kernel1D` (*model=None, x_size=None, array=None, **kwargs*)
Bases: `astropy.convolution.Kernel`

Base class for 1D filter kernels.

Parameters

- model** : `FittableModel`
Model to be evaluated.
- x_size** : odd int, optional
Size of the kernel array. Default = 8 * width.
- array** : `ndarray`
Kernel array.
- width** : number
Width of the filter kernel.
- mode** : str, optional

One of the following discretization modes:

- **‘center’ (default)**
Discretize model by taking the value at the center of the bin.
- **‘linear_interp’**
Discretize model by linearly interpolating between the values at the corners of the bin.
- **‘oversample’**
Discretize model by taking the average on an oversampled grid.
- **‘integrate’**
Discretize model by integrating the model over the bin.

- factor** : number, optional
Factor of oversampling. Default factor = 10.

Kernel2D

class `astropy.convolution.Kernel2D` (*model=None, x_size=None, y_size=None, array=None, **kwargs*)
Bases: `astropy.convolution.Kernel`

Bases: `astropy.convolution.Kernel`

Base class for 2D filter kernels.

Parameters**model** : `FittableModel`

Model to be evaluated.

x_size : odd int, optional

Size in x direction of the kernel array. Default = 8 * width.

y_size : odd int, optional

Size in y direction of the kernel array. Default = 8 * width.

array : ndarray

Kernel array.

mode : str, optional**One of the following discretization modes:**•**‘center’ (default)**

Discretize model by taking the value at the center of the bin.

•**‘linear_interp’**

Discretize model by performing a bilinear interpolation between the values at the corners of the bin.

•**‘oversample’**

Discretize model by taking the average on an oversampled grid.

•**‘integrate’**

Discretize model by integrating the model over the bin.

width : number

Width of the filter kernel.

factor : number, optional

Factor of oversampling. Default factor = 10.

MexicanHat1DKernel**class** `astropy.convolution.MexicanHat1DKernel` (*width*, ***kwargs*)Bases: `astropy.convolution.Kernel1D`

1D Mexican hat filter kernel.

The Mexican Hat, or inverted Gaussian-Laplace filter, is a bandpass filter. It smoothes the data and removes slowly varying or constant structures (e.g. Background). It is useful for peak or multi-scale detection.

This kernel is derived from a normalized Gaussian function, by computing the second derivative. This results in an amplitude at the kernels center of $1. / (\text{sqrt}(2 * \text{pi}) * \text{width} ** 3)$. The normalization is the same as for `scipy.ndimage.filters.gaussian_laplace`, except for a minus sign.

Parameters**width** : number

Width of the filter kernel, defined as the standard deviation of the Gaussian function from which it is derived.

x_size : odd int, optional

Size in x direction of the kernel array. Default = $8 * \text{width}$.

mode : str, optional

One of the following discretization modes:

- ‘center’ (default)**

Discretize model by taking the value at the center of the bin.

- ‘linear_interp’**

Discretize model by linearly interpolating between the values at the corners of the bin.

- ‘oversample’**

Discretize model by taking the average on an oversampled grid.

- ‘integrate’**

Discretize model by integrating the model over the bin.

factor : number, optional

Factor of oversampling. Default factor = 10.

See also:

`Box1DKernel`, `Gaussian1DKernel`, `Trapezoid1DKernel`

Examples

Kernel response:

```
import matplotlib.pyplot as plt
from astropy.convolution import MexicanHat1DKernel
mexicanhat_1d_kernel = MexicanHat1DKernel(10)
plt.plot(mexicanhat_1d_kernel, drawstyle='steps')
plt.xlabel('x [pixels]')
plt.ylabel('value')
plt.show()
```

MexicanHat2DKernel

class `astropy.convolution.MexicanHat2DKernel` (*width*, ***kwargs*)

Bases: `astropy.convolution.Kernel2D`

2D Mexican hat filter kernel.

The Mexican Hat, or inverted Gaussian-Laplace filter, is a bandpass filter. It smoothes the data and removes slowly varying or constant structures (e.g. Background). It is useful for peak or multi-scale detection.

This kernel is derived from a normalized Gaussian function, by computing the second derivative. This results in an amplitude at the kernels center of $1. / (\pi * \text{width} ** 4)$. The normalization is the same as for `scipy.ndimage.filters.gaussian_laplace`, except for a minus sign.

Parameters

width : number

Width of the filter kernel, defined as the standard deviation of the Gaussian function from which it is derived.

x_size : odd int, optional

Size in x direction of the kernel array. Default = 8 * width.

y_size : odd int, optional

Size in y direction of the kernel array. Default = 8 * width.

mode : str, optional

One of the following discretization modes:

- ‘center’ (default)**

Discretize model by taking the value at the center of the bin.

- ‘linear_interp’**

Discretize model by performing a bilinear interpolation between the values at the corners of the bin.

- ‘oversample’**

Discretize model by taking the average on an oversampled grid.

- ‘integrate’**

Discretize model by integrating the model over the bin.

factor : number, optional

Factor of oversampling. Default factor = 10.

See also:

`Box2DKernel`, `Tophat2DKernel`, `MexicanHat2DKernel`, `Ring2DKernel`,
`TrapezoidDisk2DKernel`, `AiryDisk2DKernel`

Examples

Kernel response:

```
import matplotlib.pyplot as plt
from astropy.convolution import MexicanHat2DKernel
mexicanhat_2D_kernel = MexicanHat2DKernel(10)
plt.imshow(mexicanhat_2D_kernel, interpolation='none', origin='lower')
plt.xlabel('x [pixels]')
plt.ylabel('y [pixels]')
plt.colorbar()
plt.show()
```

Model1DKernel

class `astropy.convolution.Model1DKernel` (*model*, ****kwargs**)

Bases: `astropy.convolution.Kernel1D`

Create kernel from 1D model.

The model has to be centered on $x = 0$.

Parameters

model : `Fittable1DModel`

Kernel response function model

x_size : odd int, optional

Size in x direction of the kernel array. Default = 8 * width.

mode : str, optional

One of the following discretization modes:

- ‘center’ (default)**

Discretize model by taking the value at the center of the bin.

- ‘linear_interp’**

Discretize model by linearly interpolating between the values at the corners of the bin.

- ‘oversample’**

Discretize model by taking the average on an oversampled grid.

- ‘integrate’**

Discretize model by integrating the model over the bin.

factor : number, optional

Factor of oversampling. Default factor = 10.

Raises

TypeError

If model is not an instance of `Fittable1DModel`

See also:

Model2DKernel

Create kernel from `Fittable2DModel`

CustomKernel

Create kernel from list or array

Examples

Define a Gaussian1D model:

```
>>> from astropy.modeling.models import Gaussian1D
>>> from astropy.convolution.kernels import Model1DKernel
>>> gauss = Gaussian1D(1, 0, 2)
```

And create a custom one dimensional kernel from it:

```
>>> gauss_kernel = Model1DKernel(gauss, x_size=9)
```

This kernel can now be used like a usual Astropy kernel.

Model2DKernel

class `astropy.convolution.Model2DKernel` (*model*, ****kwargs**)

Bases: `astropy.convolution.Kernel2D`

Create kernel from 2D model.

The model has to be centered on $x = 0$ and $y = 0$.

Parameters**model** : `Fittable2DModel`

Kernel response function model

x_size : odd int, optional

Size in x direction of the kernel array. Default = 8 * width.

y_size : odd int, optional

Size in y direction of the kernel array. Default = 8 * width.

mode : str, optional**One of the following discretization modes:**•**‘center’ (default)**

Discretize model by taking the value at the center of the bin.

•**‘linear_interp’**

Discretize model by performing a bilinear interpolation between the values at the corners of the bin.

•**‘oversample’**

Discretize model by taking the average on an oversampled grid.

•**‘integrate’**

Discretize model by integrating the model over the bin.

factor : number, optional

Factor of oversampling. Default factor = 10.

Raises**TypeError**If model is not an instance of `Fittable2DModel`**See also:****`Model1DKernel`**Create kernel from `Fittable1DModel`**`CustomKernel`**

Create kernel from list or array

Examples

Define a Gaussian2D model:

```
>>> from astropy.modeling.models import Gaussian2D
>>> from astropy.convolution.kernels import Model2DKernel
>>> gauss = Gaussian2D(1, 0, 0, 2, 2)
```

And create a custom two dimensional kernel from it:

```
>>> gauss_kernel = Model2DKernel(gauss, x_size=9)
```

This kernel can now be used like a usual astropy kernel.

Ring2DKernel

class `astropy.convolution.Ring2DKernel` (*radius_in*, *width*, ****kwargs**)

Bases: `astropy.convolution.Kernel2D`

2D Ring filter kernel.

The Ring filter kernel is the difference between two Tophat kernels of different width. This kernel is useful for, e.g., background estimation.

Parameters

radius_in : number

Inner radius of the ring kernel.

width : number

Width of the ring kernel.

mode: str, optional

One of the following discretization modes:

• **‘center’ (default)**

Discretize model by taking the value at the center of the bin.

• **‘linear_interp’**

Discretize model by performing a bilinear interpolation between the values at the corners of the bin.

• **‘oversample’**

Discretize model by taking the average on an oversampled grid.

• **‘integrate’**

Discretize model by integrating the model over the bin.

factor : number, optional

Factor of oversampling. Default factor = 10.

See also:

`Box2DKernel`, `Gaussian2DKernel`, `MexicanHat2DKernel`, `Tophat2DKernel`

Examples

Kernel response:

```
import matplotlib.pyplot as plt
from astropy.convolution import Ring2DKernel
ring_2D_kernel = Ring2DKernel(9, 8)
plt.imshow(ring_2D_kernel, interpolation='none', origin='lower')
plt.xlabel('x [pixels]')
plt.ylabel('y [pixels]')
plt.colorbar()
plt.show()
```

Tophat2DKernel

class `astropy.convolution.Tophat2DKernel` (*radius*, ***kwargs*)

Bases: `astropy.convolution.Kernel2D`

2D Tophat filter kernel.

The Tophat filter is an isotropic smoothing filter. It can produce artifacts when applied repeatedly on the same data.

Parameters

radius : int

Radius of the filter kernel.

mode : str, optional

One of the following discretization modes:

- **‘center’ (default)**

Discretize model by taking the value at the center of the bin.

- **‘linear_interp’**

Discretize model by performing a bilinear interpolation between the values at the corners of the bin.

- **‘oversample’**

Discretize model by taking the average on an oversampled grid.

- **‘integrate’**

Discretize model by integrating the model over the bin.

factor : number, optional

Factor of oversampling. Default factor = 10.

See also:

`Box2DKernel`, `Tophat2DKernel`, `MexicanHat2DKernel`, `Ring2DKernel`, `TrapezoidDisk2DKernel`, `AiryDisk2DKernel`

Examples

Kernel response:

```
import matplotlib.pyplot as plt
from astropy.convolution import Tophat2DKernel
tophat_2d_kernel = Tophat2DKernel(40)
plt.imshow(tophat_2d_kernel, interpolation='none', origin='lower')
plt.xlabel('x [pixels]')
plt.ylabel('y [pixels]')
plt.colorbar()
plt.show()
```

Trapezoid1DKernel

class `astropy.convolution.Trapezoid1DKernel` (*width*, *slope=1.0*, ***kwargs*)

Bases: `astropy.convolution.Kernel1D`

1D trapezoid kernel.

Parameters

width : number

Width of the filter kernel, defined as the width of the constant part, before it begins to slope down.

slope : number

Slope of the filter kernel's tails

mode : str, optional

One of the following discretization modes:

- **'center' (default)**

Discretize model by taking the value at the center of the bin.

- **'linear_interp'**

Discretize model by linearly interpolating between the values at the corners of the bin.

- **'oversample'**

Discretize model by taking the average on an oversampled grid.

- **'integrate'**

Discretize model by integrating the model over the bin.

factor : number, optional

Factor of oversampling. Default factor = 10.

See also:

[Box1DKernel](#), [Gaussian1DKernel](#), [MexicanHat1DKernel](#)

Examples

Kernel response:

```
import matplotlib.pyplot as plt
from astropy.convolution import Trapezoid1DKernel
trapezoid_1d_kernel = Trapezoid1DKernel(17, slope=0.2)
plt.plot(trapezoid_1d_kernel, drawstyle='steps')
plt.xlabel('x [pixels]')
plt.ylabel('amplitude')
plt.xlim(-1, 28)
plt.show()
```

TrapezoidDisk2DKernel

class `astropy.convolution.TrapezoidDisk2DKernel` (*radius, slope=1.0, **kwargs*)

Bases: `astropy.convolution.Kernel2D`

2D trapezoid kernel.

Parameters

radius : number

Width of the filter kernel, defined as the width of the constant part, before it begins to slope down.

slope : number

Slope of the filter kernel's tails

mode : str, optional

One of the following discretization modes:

- **'center' (default)**

Discretize model by taking the value at the center of the bin.

- **'linear_interp'**

Discretize model by performing a bilinear interpolation between the values at the corners of the bin.

- **'oversample'**

Discretize model by taking the average on an oversampled grid.

- **'integrate'**

Discretize model by integrating the model over the bin.

factor : number, optional

Factor of oversampling. Default factor = 10.

See also:

[Box2DKernel](#), [Tophat2DKernel](#), [MexicanHat2DKernel](#), [Ring2DKernel](#),
[TrapezoidDisk2DKernel](#), [AiryDisk2DKernel](#)

Examples

Kernel response:

```
import matplotlib.pyplot as plt
from astropy.convolution import TrapezoidDisk2DKernel
trapezoid_2D_kernel = TrapezoidDisk2DKernel(20, slope=0.2)
plt.imshow(trapezoid_2D_kernel, interpolation='none', origin='lower')
plt.xlabel('x [pixels]')
plt.ylabel('y [pixels]')
plt.colorbar()
plt.show()
```

COSMOLOGICAL CALCULATIONS (ASTROPY.COSMOLOGY)

19.1 Introduction

The `astropy.cosmology` subpackage contains classes for representing cosmologies, and utility functions for calculating commonly used quantities that depend on a cosmological model. This includes distances, ages and lookback times corresponding to a measured redshift or the transverse separation corresponding to a measured angular separation.

19.2 Getting Started

Cosmological quantities are calculated using methods of a `Cosmology` object. For example, to calculate the Hubble constant at $z=0$ (i.e., H_0), and the number of transverse proper kpc corresponding to an arcminute at $z=3$:

```
>>> from astropy.cosmology import WMAP9 as cosmo
>>> cosmo.H(0)
<Quantity 69.32 km / (Mpc s)>

>>> cosmo.kpc_proper_per_arcmin(3)
<Quantity 472.97709620405266 kpc / arcmin>
```

Here `WMAP9` is a built-in object describing a cosmology with the parameters from the 9-year WMAP results. Several other built-in cosmologies are also available, see [Built-in Cosmologies](#). The available methods of the cosmology object are listed in the methods summary for the `FLRW` class. If you're using IPython you can also use tab completion to print a list of the available methods. To do this, after importing the cosmology as in the above example, type `cosmo.` at the IPython prompt and then press the tab key.

All of these methods also accept an array of redshifts as input:

```
>>> from astropy.cosmology import WMAP9 as cosmo
>>> cosmo.comoving_distance([0.5, 1.0, 1.5])
<Quantity [ 1916.0694236 , 3363.07064333, 4451.74756242] Mpc>
```

You can create your own arbitrary cosmology using one of the `Cosmology` classes:

```
>>> from astropy.cosmology import FlatLambdaCDM
>>> cosmo = FlatLambdaCDM(H0=70, Om0=0.3)
>>> cosmo
FlatLambdaCDM(H0=70 km / (Mpc s), Om0=0.3, Tcmb0=2.725 K,
              Neff=3.04, m_nu=[ 0.  0.  0.] eV)
```

The cosmology subpackage makes use of `units`, so in many cases returns values with units attached. Consult the documentation for that subpackage for more details, but briefly, to access the floating point or array values:

```
>>> from astropy.cosmology import WMAP9 as cosmo
>>> H0 = cosmo.H(0)
>>> H0.value, H0.unit
(69.32, Unit("km / (Mpc s)"))
```

19.3 Using `astropy.cosmology`

Most of the functionality is enabled by the `FLRW` object. This represents a homogeneous and isotropic cosmology (characterized by the Friedmann-Lemaître-Robertson-Walker metric, named after the people who solved Einstein's field equation for this special case). However, you can't work with this class directly, as you must specify a dark energy model by using one of its subclasses instead, such as `FlatLambdaCDM`.

You can create a new `FlatLambdaCDM` object with arguments giving the Hubble parameter and omega matter (both at $z=0$):

```
>>> from astropy.cosmology import FlatLambdaCDM
>>> cosmo = FlatLambdaCDM(H0=70, Om0=0.3)
>>> cosmo
FlatLambdaCDM(H0=70 km / (Mpc s), Om0=0.3, Tcmb0=2.725 K,
              Neff=3.04, m_nu=[ 0.  0.  0.] eV)
```

This can also be done more explicitly using units, which is recommended:

```
>>> from astropy.cosmology import FlatLambdaCDM
>>> import astropy.units as u
>>> cosmo = FlatLambdaCDM(H0=70 * u.km / u.s / u.Mpc, Om0=0.3)
```

However, most of the parameters that accept units (`H0`, `Tcmb0`) have default units, so unit quantities do not have to be used. The exception are neutrino masses, where you must supply a units if you want massive neutrinos.

The pre-defined cosmologies described in the [Getting Started](#) section are instances of `FlatLambdaCDM`, and have the same methods. So we can find the luminosity distance to redshift 4 by:

```
>>> cosmo.luminosity_distance(4)
<Quantity 35842.353618623194 Mpc>
```

or the age of the universe at $z = 0$:

```
>>> cosmo.age(0)
<Quantity 13.461701658024014 Gyr>
```

They also accept arrays of redshifts:

```
>>> cosmo.age([0.5, 1, 1.5]).value
array([ 8.42128047,  5.74698053,  4.19645402])
```

See the `FLRW` and `FlatLambdaCDM` object docstring for all the methods and attributes available. In addition to flat Universes, non-flat varieties are supported such as `LambdaCDM`. There are also a variety of standard cosmologies with the parameters already defined (see [Built-in Cosmologies](#)):

```
>>> from astropy.cosmology import WMAP7 # WMAP 7-year cosmology
>>> WMAP7.critical_density(0) # critical density at z = 0
<Quantity 9.31000324385361e-30 g / cm3>
```

You can see how the density parameters evolve with redshift as well:

```
>>> from astropy.cosmology import WMAP7 # WMAP 7-year cosmology
>>> WMAP7.Om([0, 1.0, 2.0]), WMAP7.Ode([0., 1.0, 2.0])
(array([ 0.272      ,  0.74898524,  0.90905239]),
 array([ 0.72791572,  0.25055061,  0.0901026 ]))
```

Note that these don't quite add up to one even though WMAP7 assumes a flat Universe because photons and neutrinos are included. Also note that they are unitless and so are not `Quantity` objects.

Cosmological instances have an optional `name` attribute which can be used to describe the cosmology:

```
>>> from astropy.cosmology import FlatwCDM
>>> cosmo = FlatwCDM(name='SNLS3+WMAP7', H0=71.58, Om0=0.262, w0=-1.016)
>>> cosmo
FlatwCDM(name="SNLS3+WMAP7", H0=71.6 km / (Mpc s), Om0=0.262,
          w0=-1.02, Tcmb0=2.725 K, Neff=3.04, m_nu=[ 0.  0.  0.] eV)
```

This is also an example with a different model for dark energy, a flat Universe with a constant dark energy equation of state, but not necessarily a cosmological constant. A variety of additional dark energy models are also supported – see [Specifying a dark energy model](#).

A important point is that the cosmological parameters of each instance are immutable – that is, if you want to change, say, `Om`, you need to make a new instance of the class.

19.3.1 Finding the Redshift at a Given Value of a Cosmological Quantity

If you know a cosmological quantity and you want to know the redshift which it corresponds to, you can use `z_at_value`:

```
>>> import astropy.units as u
>>> from astropy.cosmology import Planck13, z_at_value
>>> z_at_value(Planck13.age, 2 * u.Gyr)
3.1981226843560968
```

For some quantities there can be more than one redshift that satisfies a value. In this case you can use the `zmin` and `zmax` keywords to restrict the search range. See the `z_at_value` docstring for more detailed usage examples.

19.3.2 Built-in Cosmologies

A number of pre-loaded cosmologies are available from analyses using the WMAP and Planck satellite data. For example,

```
>>> from astropy.cosmology import Planck13 # Planck 2013
>>> Planck13.lookback_time(2) # lookback time in Gyr at z=2
<Quantity 10.511841788576083 Gyr>
```

A full list of the pre-defined cosmologies is given by `cosmology.parameters.available`, and summarized below:

Name	Source	H0	Om	Flat
WMAP5	Komatsu et al. 2009	70.2	0.277	Yes
WMAP7	Komatsu et al. 2011	70.4	0.272	Yes
WMAP9	Hinshaw et al. 2013	69.3	0.287	Yes
Planck13	Planck Collab 2013, Paper XVI	67.8	0.307	Yes

Currently, all are instances of `FlatLambdaCDM`. More details about exactly where each set of parameters come from are available in the docstring for each object:

```
>>> from astropy.cosmology import WMAP7
>>> print(WMAP7.__doc__)
WMAP7 instance of FlatLambdaCDM cosmology
(from Komatsu et al. 2011, ApJS, 192, 18, doi: 10.1088/0067-0049/192/2/18.
Table 1 (WMAP + BAO + H0 ML).)
```

19.3.3 Specifying a dark energy model

In addition to the standard `FlatLambdaCDM` model described above, a number of additional dark energy models are provided. `FlatLambdaCDM` and `LambdaCDM` assume that dark energy is a cosmological constant, and should be the most commonly used cases; the former assumes a flat Universe, the latter allows for spatial curvature. `FlatwCDM` and `wCDM` assume a constant dark energy equation of state parameterized by w_0 . Two forms of a variable dark energy equation of state are provided: the simple first order linear expansion $w(z) = w_0 + w_z z$ by `w0wzCDM`, as well as the common CPL form by `w0waCDM`: $w(z) = w_0 + w_a(1 - a) = w_0 + w_a z / (1 + z)$ and its generalization to include a pivot redshift by `wpwaCDM`: $w(z) = w_p + w_a(a_p - a)$.

Users can specify their own equation of state by sub-classing `FLRW`. See the provided subclasses for examples.

19.3.4 Photons and Neutrinos

The cosmology classes include the contribution to the energy density from both photons and neutrinos. By default, the latter are assumed massless. The three parameters controlling the properties of these species, which are arguments to the initializers of all the cosmological classes, are `Tcmb0` (the temperature of the CMB at $z=0$), `Neff`, the effective number of neutrino species, and `m_nu`, the rest mass of the neutrino species. `Tcmb0` and `m_nu` should be expressed as unit Quantities. All three have standard default values (2.725 K, 3.04, and 0 eV respectively; the reason that `Neff` is not 3 primarily has to do with a small bump in the neutrino energy spectrum due to electron-positron annihilation, but is also affected by weak interaction physics).

Massive neutrinos are treated using the approach described in the WMAP 7-year cosmology paper (Komatsu et al. 2011, ApJS, 192, 18, section 3.3). This is not the simple $\Omega_{\nu} h^2 = \sum_i m_{\nu i} / 93.04 \text{ eV}$ approximation. Also note that the values of $\Omega_{\nu}(z)$ include both the kinetic energy and the rest-mass energy components, and that the Planck13 cosmology includes a single species of neutrinos with non-zero mass (which is not included in Ω_{m0}).

The contribution of photons and neutrinos to the total mass-energy density can be found as a function of redshift:

```
>>> from astropy.cosmology import WMAP7 # WMAP 7-year cosmology
>>> WMAP7.Ogamma0, WMAP7.Onu0 # Current epoch values
(4.985694972799396e-05, 3.442154948307989e-05)
>>> z = [0, 1.0, 2.0]
>>> WMAP7.Ogamma(z), WMAP7.Onu(z)
(array([ 4.98569497e-05,  2.74574409e-04,  4.99881391e-04]),
 array([ 3.44215495e-05,  1.89567887e-04,  3.45121234e-04]))
```

If you want to exclude photons and neutrinos from your calculations, simply set `Tcmb0` to 0:

```
>>> from astropy.cosmology import FlatLambdaCDM
>>> import astropy.units as u
>>> cos = FlatLambdaCDM(70.4 * u.km / u.s / u.Mpc, 0.272, Tcmb0 = 0.0 * u.K)
>>> cos.Ogamma0, cos.Onu0
(0.0, 0.0)
```

Neutrinos can be removed (while leaving photons) by setting `Neff` to 0:

```
>>> from astropy.cosmology import FlatLambdaCDM
>>> cos = FlatLambdaCDM(70.4, 0.272, Neff=0)
>>> cos.Ogamma([0, 1, 2]) # Photons are still present
```

```
array([ 4.98569497e-05,  2.74623215e-04,  5.00051839e-04])
>>> cos.Onu([0, 1, 2]) # But not neutrinos
array([ 0.,  0.,  0.])
```

The number of neutrino species is assumed to be the floor of N_{eff} , which in the default case is 3. Therefore, if non-zero neutrino masses are desired, then 3 masses should be provided. However, if only one value is provided, all the species are assumed to have the same mass. N_{eff} is assumed to be shared equally between each species.

```
>>> from astropy.cosmology import FlatLambdaCDM
>>> import astropy.units as u
>>> H0 = 70.4 * u.km / u.s / u.Mpc
>>> m_nu = 0 * u.eV
>>> cosmo = FlatLambdaCDM(H0, 0.272, m_nu=m_nu)
>>> cosmo.has_massive_nu
False
>>> cosmo.m_nu
<Quantity [ 0., 0., 0.] eV>
>>> m_nu = [0.0, 0.05, 0.10] * u.eV
>>> cosmo = FlatLambdaCDM(H0, 0.272, m_nu=m_nu)
>>> cosmo.has_massive_nu
True
>>> cosmo.m_nu
<Quantity [ 0. , 0.05, 0.1 ] eV>
>>> cosmo.Onu([0, 1.0, 15.0])
array([ 0.00326988,  0.00896783,  0.0125786 ])
>>> cosmo.Onu(1) * cosmo.critical_density(1)
<Quantity 2.444380380370406e-31 g / cm3>
```

While these examples used `FlatLambdaCDM`, the above examples also apply for all of the other cosmology classes.

19.4 For Developers: Using `astropy.cosmology` inside Astropy

If you are writing code for the Astropy core or an affiliated package, it’s often useful to assume a default cosmology, so that the exact cosmology doesn’t have to be specified every time a function or method is called. In this case it’s possible to specify a “default” cosmology.

You can set the default cosmology to a pre-defined value by using the “`default_cosmology`” option in the `[cosmology.core]` section of the configuration file (see *Configuration system (astropy.config)*). Alternatively, you can use the `set` function of `default_cosmology` to set a cosmology for the current Python session. If you haven’t set a default cosmology using one of the methods described above, then the cosmology module will default to using the 9-year WMAP parameters.

It is strongly recommended that you use the default cosmology through the `default_cosmology` science state object. An override option can then be provided using something like the following:

```
def myfunc(..., cosmo=None):
    from astropy.cosmology import default_cosmology

    if cosmo is None:
        cosmo = default_cosmology.get()

    ... your code here ...
```

This ensures that all code consistently uses the default cosmology unless explicitly overridden.

Note: In general it’s better to use an explicit cosmology (for example `WMAP9.H(0)`) instead of

`cosmology.default_cosmology.get().H(0)`). Use of the default cosmology should generally be reserved for code that will be included in the Astropy core or an affiliated package.

19.5 See Also

- Hogg, “Distance measures in cosmology”, <http://arxiv.org/abs/astro-ph/9905116>
- Linder, “Exploring the Expansion History of the Universe”, <http://arxiv.org/abs/astro-ph/0208512>
- NASA’s Legacy Archive for Microwave Background Data Analysis, <http://lambda.gsfc.nasa.gov/>

19.6 Range of validity and reliability

The code in this sub-package is tested against several widely-used online cosmology calculators, and has been used to perform many calculations in refereed papers. You can check the range of redshifts over which the code is regularly tested in the module `astropy.cosmology.tests.test_cosmology`. If you find any bugs, please let us know by [opening an issue at the github repository!](#)

The built in cosmologies use the parameters as listed in the respective papers. These provide only a limited range of precision, and so you should not expect derived quantities to match beyond that precision. For example, the Planck 2013 results only provide the Hubble constant to 4 digits. Therefore, the Planck13 built-in cosmology should only be expected to match the age of the Universe quoted by the Planck team to 4 digits, although they provide 5 in the paper.

19.7 Reference/API

19.7.1 `astropy.cosmology` Module

`astropy.cosmology` contains classes and functions for cosmological distance measures and other cosmology-related calculations.

See the [Astropy documentation](#) for more detailed usage examples and references.

Functions

<code>H(*args, **kwargs)</code>	Deprecated since version 0.4.
<code>angular_diameter_distance(*args, **kwargs)</code>	Deprecated since version 0.4.
<code>arcsec_per_kpc_comoving(*args, **kwargs)</code>	Deprecated since version 0.4.
<code>arcsec_per_kpc_proper(*args, **kwargs)</code>	Deprecated since version 0.4.
<code>comoving_distance(*args, **kwargs)</code>	Deprecated since version 0.4.
<code>critical_density(*args, **kwargs)</code>	Deprecated since version 0.4.
<code>distmod(*args, **kwargs)</code>	Deprecated since version 0.4.
<code>get_current(*args, **kwargs)</code>	Deprecated since version 0.4.
<code>kpc_comoving_per_arcmin(*args, **kwargs)</code>	Deprecated since version 0.4.
<code>kpc_proper_per_arcmin(*args, **kwargs)</code>	Deprecated since version 0.4.
<code>lookback_time(*args, **kwargs)</code>	Deprecated since version 0.4.
<code>luminosity_distance(*args, **kwargs)</code>	Deprecated since version 0.4.
<code>scale_factor(*args, **kwargs)</code>	Deprecated since version 0.4.
<code>set_current(*args, **kwargs)</code>	Deprecated since version 0.4.

Continued on next page

Table 19.1 – continued from previous page

<code>z_at_value(func, fval[, zmin, zmax, ztol, ...])</code>	Find the redshift z at which $func(z) = fval$.
--	---

H

`astropy.cosmology.H(*args, **kwargs)`

Deprecated since version 0.4: The H function is deprecated and may be removed in a future version. Use `<Cosmology object>.H` instead.

Hubble parameter (km/s/Mpc) at redshift z .

Parameters

z : array_like

Input redshifts.

Returns

H : Quantity

Hubble parameter at each input redshift.

angular_diameter_distance

`astropy.cosmology.angular_diameter_distance(*args, **kwargs)`

Deprecated since version 0.4: The `angular_diameter_distance` function is deprecated and may be removed in a future version. Use `<Cosmology object>.angular_diameter_distance` instead.

Angular diameter distance in Mpc at a given redshift.

This gives the proper (sometimes called ‘physical’) transverse distance corresponding to an angle of 1 radian for an object at redshift z .

Parameters

z : array_like

Input redshifts.

Returns

angdist : Quantity

Angular diameter distance at each input redshift.

arcsec_per_kpc_comoving

`astropy.cosmology.arcsec_per_kpc_comoving(*args, **kwargs)`

Deprecated since version 0.4: The `arcsec_per_kpc_comoving` function is deprecated and may be removed in a future version. Use `<Cosmology object>.arcsec_per_kpc_comoving` instead.

Angular separation in arcsec corresponding to a comoving kpc
at redshift z .

Parameters

z : array_like

Input redshifts.

Returns

theta : Quantity

The angular separation in arcsec corresponding to a comoving kpc at each input redshift.

`arcsec_per_kpc_proper`

`astropy.cosmology.arcsec_per_kpc_proper(*args, **kwargs)`

Deprecated since version 0.4: The `arcsec_per_kpc_proper` function is deprecated and may be removed in a future version. Use `<Cosmology object>.arcsec_per_kpc_proper` instead.

Angular separation in arcsec corresponding to a proper kpc at redshift z .

Parameters

`z` : array_like

Input redshifts.

Returns

`theta` : Quantity

The angular separation in arcsec corresponding to a proper kpc at each input redshift.

`comoving_distance`

`astropy.cosmology.comoving_distance(*args, **kwargs)`

Deprecated since version 0.4: The `comoving_distance` function is deprecated and may be removed in a future version. Use `<Cosmology object>.comoving_distance` instead.

Comoving distance in Mpc at redshift z .

The comoving distance along the line-of-sight between two objects remains constant with time for objects in the Hubble flow.

Parameters

`z` : array_like

Input redshifts.

Returns

`codist` : Quantity

Comoving distance at each input redshift.

`critical_density`

`astropy.cosmology.critical_density(*args, **kwargs)`

Deprecated since version 0.4: The `critical_density` function is deprecated and may be removed in a future version. Use `<Cosmology object>.critical_density` instead.

Critical density in grams per cubic cm at redshift z .

Parameters

`z` : array_like

Input redshifts.

Returns

`critdens` : Quantity

Critical density at each input redshift.

distmod

`astropy.cosmology.distmod(*args, **kwargs)`

Deprecated since version 0.4: The `distmod` function is deprecated and may be removed in a future version. Use `<Cosmology object>.distmod` instead.

Distance modulus at redshift z .

The distance modulus is defined as the (apparent magnitude - absolute magnitude) for an object at redshift z .

Parameters

z : array_like

Input redshifts.

Returns

distmod : Quantity

Distance modulus at each input redshift.

See also:

[z_at_value](#)

Find the redshift corresponding to a distance modulus.

get_current

`astropy.cosmology.get_current(*args, **kwargs)`

Deprecated since version 0.4: The `get_current` function is deprecated and may be removed in a future version. Use `astropy.cosmology.default_cosmology.get` instead.

Get the current cosmology.

If no current has been set, the WMAP9 cosmology is returned and a warning is given.

Returns

cosmo : Cosmology instance

kpc_comoving_per_arcmin

`astropy.cosmology.kpc_comoving_per_arcmin(*args, **kwargs)`

Deprecated since version 0.4: The `kpc_comoving_per_arcmin` function is deprecated and may be removed in a future version. Use `<Cosmology object>.kpc_comoving_per_arcmin` instead.

Separation in transverse comoving kpc corresponding to an arcminute at redshift z .

Parameters

z : array_like

Input redshifts.

Returns

d: `Quantity`

The distance in comoving kpc corresponding to an arcmin at each input redshift.

kpc_proper_per_arcmin

`astropy.cosmology.kpc_proper_per_arcmin(*args, **kwargs)`

Deprecated since version 0.4: The `kpc_proper_per_arcmin` function is deprecated and may be removed in a future version. Use `<Cosmology object>.kpc_proper_per_arcmin` instead.

Separation in transverse proper kpc corresponding to an arcminute at redshift z.

Parameters

z: `array_like`

Input redshifts.

Returns

d: `Quantity`

The distance in proper kpc corresponding to an arcmin at each input redshift.

lookback_time

`astropy.cosmology.lookback_time(*args, **kwargs)`

Deprecated since version 0.4: The `lookback_time` function is deprecated and may be removed in a future version. Use `<Cosmology object>.lookback_time` instead.

Lookback time in Gyr to redshift z.

The lookback time is the difference between the age of the Universe now and the age at redshift z.

Parameters

z: `array_like`

Input redshifts.

Returns

t: `Quantity`

Lookback time at each input redshift.

See also:

`z_at_value`

Find the redshift corresponding to a lookback time.

luminosity_distance

`astropy.cosmology.luminosity_distance(*args, **kwargs)`

Deprecated since version 0.4: The `luminosity_distance` function is deprecated and may be removed in a future version. Use `<Cosmology object>.luminosity_distance` instead.

Luminosity distance in Mpc at redshift z.

This is the distance to use when converting between the bolometric flux from an object at redshift z and its bolometric luminosity.

Parameters

z : array_like

Input redshifts.

Returns

lumdist : Quantity

Luminosity distance at each input redshift.

See also:

[z_at_value](#)

Find the redshift corresponding to a luminosity distance.

scale_factor

`astropy.cosmology.scale_factor(*args, **kwargs)`

Deprecated since version 0.4: The `scale_factor` function is deprecated and may be removed in a future version. Use `<Cosmology object>.scale_factor` instead.

Scale factor at redshift z .

The scale factor is defined as $a = 1 / (1 + z)$.

Parameters

z : array_like

Input redshifts.

Returns

scalefac : ndarray, or float if input scalar

Scale factor at each input redshift.

set_current

`astropy.cosmology.set_current(*args, **kwargs)`

Deprecated since version 0.4: The `set_current` function is deprecated and may be removed in a future version. Use `astropy.cosmology.default_cosmology.set` instead.

Set the current cosmology.

Call this with an empty string ('') to get a list of the strings that map to available pre-defined cosmologies.

Parameters

cosmo : str or `Cosmology` instance

The cosmology to use.

z_at_value

`astropy.cosmology.z_at_value` (*func*, *fval*, *zmin*=0, *zmax*=1000, *ztol*=1.0000000000000001e-05, *maxfun*=500)

Find the redshift z at which $\text{func}(z) = \text{fval}$.

This finds the redshift at which one of the cosmology functions or methods (for example `Planck13.distmod`) is equal to a known value.

Warning: Make sure you understand the behaviour of the function that you are trying to invert! Depending on the cosmology, there may not be a unique solution. For example, in the standard Lambda CDM cosmology, there are two redshifts which give an angular diameter distance of 1500 Mpc, $z \sim 0.7$ and $z \sim 3.8$. To force `z_at_value` to find the solution you are interested in, use the `zmin` and `zmax` keywords to limit the search range (see the example below).

Parameters

func : function or method

A function that takes a redshift as input.

fval : `astropy.Quantity` instance

The value of $\text{func}(z)$.

zmin : float, optional

The lower search limit for z (default 0).

zmax : float, optional

The upper search limit for z (default 1000).

ztol : float, optional

The relative error in z acceptable for convergence.

maxfun : int, optional

The maximum number of function evaluations allowed in the optimization routine (default 500).

Returns

z : float

The redshift z satisfying $z_{\text{min}} < z < z_{\text{max}}$ and $\text{func}(z) = \text{fval}$ within `ztol`.

Notes

This works for any arbitrary input cosmology, but is inefficient if you want to invert a large number of values for the same cosmology. In this case, it is faster to instead generate an array of values at many closely-spaced redshifts that cover the relevant redshift range, and then use interpolation to find the redshift at each value you're interested in. For example, to efficiently find the redshifts corresponding to 10^6 values of the distance modulus in a Planck13 cosmology, you could do the following:

```
>>> import astropy.units as u
>>> from astropy.cosmology import Planck13, z_at_value
```

Generate 10^6 distance moduli between 23 and 43 for which we want to find the corresponding redshifts:

```
>>> Dvals = (23 + np.random.rand(1e6) * 20) * u.mag
```

Make a grid of distance moduli covering the redshift range we need using 50 equally log-spaced values between `zmin` and `zmax`. We use log spacing to adequately sample the steep part of the curve at low distance moduli:

```
>>> zmin = z_at_value(Planck13.distmod, Dvals.min())
>>> zmax = z_at_value(Planck13.distmod, Dvals.max())
>>> zgrid = np.logspace(zmin, zmax)
>>> Dgrid = Planck13.distmod(zgrid)
```

Finally interpolate to find the redshift at each distance modulus:

```
>>> zvals = np.interp(Dvals.value, zgrid, Dgrid.value)
```

Examples

```
>>> import astropy.units as u
>>> from astropy.cosmology import Planck13, z_at_value
```

The age and lookback time are monotonic with redshift, and so a unique solution can be found:

```
>>> z_at_value(Planck13.age, 2 * u.Gyr)
3.1981191749374629
```

The angular diameter is not monotonic however, and there are two redshifts that give a value of 1500 Mpc. Use the `zmin` and `zmax` keywords to find the one you're interested in:

```
>>> z_at_value(Planck13.angular_diameter_distance, 1500 * u.Mpc, zmax=1.5)
0.68127769625288614
>>> z_at_value(Planck13.angular_diameter_distance, 1500 * u.Mpc, zmin=2.5)
3.7914918534022011
```

Also note that the luminosity distance and distance modulus (two other commonly inverted quantities) are monotonic in flat and open universes, but not in closed universes.

Classes

<code>FLRW(H0, Om0, Ode0[, Tcmb0, Neff, m_nu, name])</code>	A class describing an isotropic and homogeneous (Friedmann-Lemaitre)
<code>FlatLambdaCDM(H0, Om0[, Tcmb0, Neff, m_nu, name])</code>	FLRW cosmology with a cosmological constant and no curvature.
<code>Flatw0wacdm(H0, Om0[, w0, wa, Tcmb0, Neff, ...])</code>	FLRW cosmology with a CPL dark energy equation of state and no curv
<code>FlatwCDM(H0, Om0[, w0, Tcmb0, Neff, m_nu, name])</code>	FLRW cosmology with a constant dark energy equation of state and no
<code>LambdaCDM(H0, Om0, Ode0[, Tcmb0, Neff, ...])</code>	FLRW cosmology with a cosmological constant and curvature.
<code>default_cosmology()</code>	The default cosmology to use.
<code>w0wacdm(H0, Om0, Ode0[, w0, wa, Tcmb0, ...])</code>	FLRW cosmology with a CPL dark energy equation of state and curvat
<code>w0wzcdm(H0, Om0, Ode0[, w0, wz, Tcmb0, ...])</code>	FLRW cosmology with a variable dark energy equation of state and cur
<code>wCDM(H0, Om0, Ode0[, w0, Tcmb0, Neff, m_nu, ...])</code>	FLRW cosmology with a constant dark energy equation of state and cu
<code>wpwacdm(H0, Om0, Ode0[, wp, wa, zp, Tcmb0, ...])</code>	FLRW cosmology with a CPL dark energy equation of state, a pivot rec

FLRW

```
class astropy.cosmology.FLRW(H0, Om0, Ode0, Tcmb0=2.7250000000000001, Neff=3.04,
                             m_nu=<Quantity 0.0 eV>, name=None)
```

Bases: `astropy.cosmology.core.Cosmology`

A class describing an isotropic and homogeneous (Friedmann-Lemaitre-Robertson-Walker) cosmology.

This is an abstract base class – you can't instantiate examples of this class, but must work with one of its

subclasses such as `LambdaCDM` or `wCDM`.

Parameters

H0 : float or scalar `Quantity`

Hubble constant at $z = 0$. If a float, must be in [km/sec/Mpc]

Om0 : float

Omega matter: density of non-relativistic matter in units of the critical density at $z=0$.

Ode0 : float

Omega dark energy: density of dark energy in units of the critical density at $z=0$.

Tcmb0 : float or scalar `Quantity`

Temperature of the CMB $z=0$. If a float, must be in [K]. Default: 2.725. Setting this to zero will turn off both photons and neutrinos (even massive ones)

Neff : float

Effective number of Neutrino species. Default 3.04.

m_nu : `Quantity`

Mass of each neutrino species. If this is a scalar `Quantity`, then all neutrino species are assumed to have that mass. Otherwise, the mass of each species. The actual number of neutrino species (and hence the number of elements of `m_nu` if it is not scalar) must be the floor of `Neff`. Usually this means you must provide three neutrino masses unless you are considering something like a sterile neutrino.

name : str

Optional name for this cosmological object.

Notes

Class instances are static – you can't change the values of the parameters. That is, all of the attributes above are read only.

Attributes Summary

<code>H0</code>	Return the Hubble constant as an <code>Quantity</code> at $z=0$
<code>Neff</code>	Number of effective neutrino species
<code>Ode0</code>	Omega dark energy; dark energy density/critical density at $z=0$
<code>Ogamma0</code>	Omega gamma; the density/critical density of photons at $z=0$
<code>Ok0</code>	Omega curvature; the effective curvature density/critical density
<code>Om0</code>	Omega matter; matter density/critical density at $z=0$
<code>Onu0</code>	Omega nu; the density/critical density of neutrinos at $z=0$
<code>Tcmb0</code>	Temperature of the CMB as <code>Quantity</code> at $z=0$
<code>Tnu0</code>	Temperature of the neutrino background as <code>Quantity</code> at $z=0$
<code>critical_density0</code>	Critical density as <code>Quantity</code> at $z=0$
<code>h</code>	Dimensionless Hubble constant: $h = H_0 / 100$ [km/sec/Mpc]
<code>has_massive_nu</code>	Does this cosmology have at least one massive neutrino species?
<code>hubble_distance</code>	Hubble distance as <code>Quantity</code>
<code>hubble_time</code>	Hubble time as <code>Quantity</code>
<code>m_nu</code>	Mass of neutrino species

Methods Summary

<code>H(z)</code>	Hubble parameter (km/s/Mpc) at redshift z .
<code>Ode(z)</code>	Return the density parameter for dark energy at redshift z .
<code>Ogamma(z)</code>	Return the density parameter for photons at redshift z .
<code>Ok(z)</code>	Return the equivalent density parameter for curvature at redshift z .
<code>Om(z)</code>	Return the density parameter for non-relativistic matter at redshift z .
<code>Onu(z)</code>	Return the density parameter for massless neutrinos at redshift z .
<code>Tcmb(z)</code>	Return the CMB temperature at redshift z .
<code>Tnu(z)</code>	Return the neutrino temperature at redshift z .
<code>absorption_distance(z)</code>	Absorption distance at redshift z .
<code>age(z)</code>	Age of the universe in Gyr at redshift z .
<code>angular_diameter_distance(z)</code>	Angular diameter distance in Mpc at a given redshift.
<code>angular_diameter_distance_z1z2(z1, z2)</code>	Angular diameter distance between objects at 2 redshifts.
<code>arcsec_per_kpc_comoving(z)</code>	Angular separation in arcsec corresponding to a comoving kpc at redshift z .
<code>arcsec_per_kpc_proper(z)</code>	Angular separation in arcsec corresponding to a proper kpc at redshift z .
<code>comoving_distance(z)</code>	Comoving line-of-sight distance in Mpc at a given redshift.
<code>comoving_transverse_distance(z)</code>	Comoving transverse distance in Mpc at a given redshift.
<code>comoving_volume(z)</code>	Comoving volume in cubic Mpc at redshift z .
<code>critical_density(z)</code>	Critical density in grams per cubic cm at redshift z .
<code>de_density_scale(z)</code>	Evaluates the redshift dependence of the dark energy density.
<code>differential_comoving_volume(z)</code>	Differential comoving volume at redshift z .
<code>distmod(z)</code>	Distance modulus at redshift z .
<code>efunc(z)</code>	Function used to calculate $H(z)$, the Hubble parameter.
<code>inv_efunc(z)</code>	Inverse of <code>efunc</code> .
<code>kpc_comoving_per_arcmin(z)</code>	Separation in transverse comoving kpc corresponding to an arcminute at redshift z .
<code>kpc_proper_per_arcmin(z)</code>	Separation in transverse proper kpc corresponding to an arcminute at redshift z .
<code>lookback_time(z)</code>	Lookback time in Gyr to redshift z .
<code>luminosity_distance(z)</code>	Luminosity distance in Mpc at redshift z .
<code>nu_relative_density(z)</code>	Neutrino density function relative to the energy density in photons.
<code>scale_factor(z)</code>	Scale factor at redshift z .
<code>w(z)</code>	The dark energy equation of state.

Attributes Documentation

H0

Return the Hubble constant as an `Quantity` at $z=0$

Neff

Number of effective neutrino species

Ode0

Omega dark energy; dark energy density/critical density at $z=0$

Ogamma0

Omega gamma; the density/critical density of photons at $z=0$

Ok0

Omega curvature; the effective curvature density/critical density at $z=0$

Om0

Omega matter; matter density/critical density at $z=0$

Onu0

Omega nu; the density/critical density of neutrinos at $z=0$

Tcmb0

Temperature of the CMB as `Quantity` at $z=0$

Tnu0

Temperature of the neutrino background as `Quantity` at $z=0$

critical_density0

Critical density as `Quantity` at $z=0$

h

Dimensionless Hubble constant: $h = H_0 / 100$ [km/sec/Mpc]

has_massive_nu

Does this cosmology have at least one massive neutrino species?

hubble_distance

Hubble distance as `Quantity`

hubble_time

Hubble time as `Quantity`

m_nu

Mass of neutrino species

Methods Documentation

H(z)

Hubble parameter (km/s/Mpc) at redshift z .

Parameters

z : array_like

Input redshifts.

Returns

H : `Quantity`

Hubble parameter at each input redshift.

Ode(z)

Return the density parameter for dark energy at redshift z .

Parameters

z : array_like

Input redshifts.

Returns

Ode : ndarray, or float if input scalar

The density of non-relativistic matter relative to the critical density at each redshift.

Ogamma(z)

Return the density parameter for photons at redshift z .

Parameters

z : array_like

Input redshifts.

Returns

Ogamma : ndarray, or float if input scalar

The energy density of photons relative to the critical density at each redshift.

Ok (*z*)

Return the equivalent density parameter for curvature at redshift *z*.

Parameters

z : array_like

Input redshifts.

Returns

Ok : ndarray, or float if input scalar

The equivalent density parameter for curvature at each redshift.

Om (*z*)

Return the density parameter for non-relativistic matter at redshift *z*.

Parameters

z : array_like

Input redshifts.

Returns

Om : ndarray, or float if input scalar

The density of non-relativistic matter relative to the critical density at each redshift.

Onu (*z*)

Return the density parameter for massless neutrinos at redshift *z*.

Parameters

z : array_like

Input redshifts.

Returns

Onu : ndarray, or float if input scalar

The energy density of photons relative to the critical density at each redshift. Note that this includes their kinetic energy (if they have mass), so it is not equal to the commonly used $\sum \frac{m_\nu}{94\text{eV}}$, which does not include kinetic energy.

Tcmb (*z*)

Return the CMB temperature at redshift *z*.

Parameters

z : array_like

Input redshifts.

Returns

Tcmb : Quantity

The temperature of the CMB in K.

Tnu (*z*)

Return the neutrino temperature at redshift *z*.

Parameters

z : array_like

Input redshifts.

Returns

Tnu : Quantity

The temperature of the cosmic neutrino background in K.

absorption_distance (*z*)

Absorption distance at redshift *z*.

This is used to calculate the number of objects with some cross section of absorption and number density intersecting a sightline per unit redshift path.

Parameters

z : array_like

Input redshifts. Must be 1D or scalar.

Returns

d : float or ndarray

Absorption distance (dimensionless) at each input redshift.

References

Hogg 1999 Section 11. (astro-ph/9905116) Bahcall, John N. and Peebles, P.J.E. 1969, ApJ, 156L, 7B

age (*z*)

Age of the universe in Gyr at redshift *z*.

Parameters

z : array_like

Input redshifts. Must be 1D or scalar.

Returns

t : [Quantity](#)

The age of the universe in Gyr at each input redshift.

See also:

[z_at_value](#)

Find the redshift corresponding to an age.

angular_diameter_distance (*z*)

Angular diameter distance in Mpc at a given redshift.

This gives the proper (sometimes called ‘physical’) transverse distance corresponding to an angle of 1 radian for an object at redshift *z*.

Weinberg, 1972, pp 421-424; Weedman, 1986, pp 65-67; Peebles, 1993, pp 325-327.

Parameters

z : array_like

Input redshifts. Must be 1D or scalar.

Returns

d : [Quantity](#)

Angular diameter distance in Mpc at each input redshift.

angular_diameter_distance_z1z2 (*z1*, *z2*)

Angular diameter distance between objects at 2 redshifts. Useful for gravitational lensing.

Parameters

z1, *z2* : array_like, shape (N,)

Input redshifts. z_2 must be large than z_1 .

Returns

d : `Quantity`, shape (N,) or single if input scalar

The angular diameter distance between each input redshift pair.

Raises

CosmologyError

If ω_k is < 0 .

Notes

This method only works for flat or open curvature ($\omega_k \geq 0$).

arcsec_per_kpc_comoving (z)

Angular separation in arcsec corresponding to a comoving kpc at redshift z .

Parameters

z : array_like

Input redshifts. Must be 1D or scalar.

Returns

theta : `Quantity`

The angular separation in arcsec corresponding to a comoving kpc at each input redshift.

arcsec_per_kpc_proper (z)

Angular separation in arcsec corresponding to a proper kpc at redshift z .

Parameters

z : array_like

Input redshifts. Must be 1D or scalar.

Returns

theta : `Quantity`

The angular separation in arcsec corresponding to a proper kpc at each input redshift.

comoving_distance (z)

Comoving line-of-sight distance in Mpc at a given redshift.

The comoving distance along the line-of-sight between two objects remains constant with time for objects in the Hubble flow.

Parameters

z : array_like

Input redshifts. Must be 1D or scalar.

Returns

d : ndarray, or float if input scalar

Comoving distance in Mpc to each input redshift.

comoving_transverse_distance (z)

Comoving transverse distance in Mpc at a given redshift.

This value is the transverse comoving distance at redshift z corresponding to an angular separation of 1 radian. This is the same as the comoving distance if ω_k is zero (as in the current concordance Λ CDM model).

Parameters**z** : array_like

Input redshifts. Must be 1D or scalar.

Returns**d** : Quantity

Comoving transverse distance in Mpc at each input redshift.

Notes

This quantity also called the ‘proper motion distance’ in some texts.

comoving_volume (z)

Comoving volume in cubic Mpc at redshift z.

This is the volume of the universe encompassed by redshifts less than z. For the case of $\omega_k = 0$ it is a sphere of radius `comoving_distance` but it is less intuitive if ω_k is not 0.**Parameters****z** : array_like

Input redshifts. Must be 1D or scalar.

Returns**V** : QuantityComoving volume in Mpc^3 at each input redshift.**critical_density** (z)

Critical density in grams per cubic cm at redshift z.

Parameters**z** : array_like

Input redshifts.

Returns**rho** : QuantityCritical density in g/cm^3 at each input redshift.**de_density_scale** (z)

Evaluates the redshift dependence of the dark energy density.

Parameters**z** : array_like

Input redshifts.

Returns**I** : ndarray, or float if input scalar

The scaling of the energy density of dark energy with redshift.

NotesThe scaling factor, I, is defined by $\rho(z) = \rho_0 I$, and is given by

$$I = \exp\left(3 \int_a^1 \frac{da'}{a'} [1 + w(a')]\right)$$

It will generally be helpful for subclasses to overload this method if the integral can be done analytically for the particular dark energy equation of state that they implement.

differential_comoving_volume(*z*)

Differential comoving volume at redshift *z*.

Useful for calculating the effective comoving volume. For example, allows for integration over a comoving volume that has a sensitivity function that changes with redshift. The total comoving volume is given by integrating `differential_comoving_volume` to redshift *z* and multiplying by a solid angle.

Parameters

z : array_like

Input redshifts.

Returns

dV : Quantity

Differential comoving volume per redshift per steradian at each input redshift.

distmod(*z*)

Distance modulus at redshift *z*.

The distance modulus is defined as the (apparent magnitude - absolute magnitude) for an object at redshift *z*.

Parameters

z : array_like

Input redshifts. Must be 1D or scalar.

Returns

distmod : Quantity

Distance modulus at each input redshift, in magnitudes

See also:

z_at_value

Find the redshift corresponding to a distance modulus.

efunc(*z*)

Function used to calculate $H(z)$, the Hubble parameter.

Parameters

z : array_like

Input redshifts.

Returns

E : ndarray, or float if input scalar

The redshift scaling of the Hubble constant.

Notes

The return value, *E*, is defined such that $H(z) = H_0 E$.

It is not necessary to override this method, but if `de_density_scale` takes a particularly simple form, it may be advantageous to.

inv_efunc(*z*)

Inverse of efunc.

Parameters

z : array_like

Input redshifts.

Returns

E : ndarray, or float if input scalar

The redshift scaling of the inverse Hubble constant.

kpc_comoving_per_arcmin(*z*)

Separation in transverse comoving kpc corresponding to an arcminute at redshift *z*.

Parameters

z : array_like

Input redshifts. Must be 1D or scalar.

Returns

d : Quantity

The distance in comoving kpc corresponding to an arcmin at each input redshift.

kpc_proper_per_arcmin(*z*)

Separation in transverse proper kpc corresponding to an arcminute at redshift *z*.

Parameters

z : array_like

Input redshifts. Must be 1D or scalar.

Returns

d : Quantity

The distance in proper kpc corresponding to an arcmin at each input redshift.

lookback_time(*z*)

Lookback time in Gyr to redshift *z*.

The lookback time is the difference between the age of the Universe now and the age at redshift *z*.

Parameters

z : array_like

Input redshifts. Must be 1D or scalar

Returns

t : Quantity

Lookback time in Gyr to each input redshift.

See also:

[z_at_value](#)

Find the redshift corresponding to a lookback time.

luminosity_distance(*z*)

Luminosity distance in Mpc at redshift *z*.

This is the distance to use when converting between the bolometric flux from an object at redshift *z* and its bolometric luminosity.

Parameters**z** : array_like

Input redshifts. Must be 1D or scalar.

Returns**d** : Quantity

Luminosity distance in Mpc at each input redshift.

See also:**z_at_value**

Find the redshift corresponding to a luminosity distance.

References

Weinberg, 1972, pp 420-424; Weedman, 1986, pp 60-62.

nu_relative_density (z)

Neutrino density function relative to the energy density in photons.

Parameters**z** : array like

Redshift

Returns**f** : ndarray, or float if z is scalar

The neutrino density scaling factor relative to the density in photons at each redshift

Notes

The density in neutrinos is given by

$$\rho_\nu(a) = 0.2271 N_{eff} f(m_\nu a / T_{\nu 0}) \rho_\gamma(a)$$

where

$$f(y) = \frac{120}{7\pi^4} \int_0^\infty dx \frac{x^2 \sqrt{x^2 + y^2}}{e^x + 1}$$

assuming that all neutrino species have the same mass. If they have different masses, a similar term is calculated for each one. Note that f has the asymptotic behavior $f(0) = 1$. This method returns $0.2271f$ using an analytical fitting formula given in Komatsu et al. 2011, ApJS 192, 18.

scale_factor (z)

Scale factor at redshift z.

The scale factor is defined as $a = 1/(1 + z)$.**Parameters****z** : array_like

Input redshifts.

Returns**a** : ndarray, or float if input scalar

Scale factor at each input redshift.

w(z)

The dark energy equation of state.

Parameters**z** : array_like

Input redshifts.

Returns**w** : ndarray, or float if input scalar

The dark energy equation of state

Notes

The dark energy equation of state is defined as $w(z) = P(z)/\rho(z)$, where $P(z)$ is the pressure at redshift z and $\rho(z)$ is the density at redshift z , both in units where $c=1$.

This must be overridden by subclasses.

FlatLambdaCDM

```
class astropy.cosmology.FlatLambdaCDM(H0, Om0, Tcmb0=2.7250000000000001, Neff=3.04,
                                       m_nu=<Quantity 0.0 eV>, name=None)
```

Bases: `astropy.cosmology.LambdaCDM`

FLRW cosmology with a cosmological constant and no curvature.

This has no additional attributes beyond those of FLRW.

Parameters**H0** : float or `Quantity`Hubble constant at $z = 0$. If a float, must be in [km/sec/Mpc]**Om0** : floatOmega matter: density of non-relativistic matter in units of the critical density at $z=0$.**Tcmb0** : float or `Quantity`Temperature of the CMB $z=0$. If a float, must be in [K]. Default: 2.725.**Neff** : float

Effective number of Neutrino species. Default 3.04.

m_nu : `Quantity`

Mass of each neutrino species. If this is a scalar `Quantity`, then all neutrino species are assumed to have that mass. Otherwise, the mass of each species. The actual number of neutrino species (and hence the number of elements of `m_nu` if it is not scalar) must be the floor of `Neff`. Usually this means you must provide three neutrino masses unless you are considering something like a sterile neutrino.

name : str

Optional name for this cosmological object.

Examples

```
>>> from astropy.cosmology import FlatLambdaCDM
>>> cosmo = FlatLambdaCDM(H0=70, Om0=0.3)
```

The comoving distance in Mpc at redshift z :

```
>>> z = 0.5
>>> dc = cosmo.comoving_distance(z)
```

Methods Summary

<code>efunc(z)</code>	Function used to calculate $H(z)$, the Hubble parameter.
<code>inv_efunc(z)</code>	Function used to calculate $\frac{1}{H_z}$.

Methods Documentation

`efunc(z)`

Function used to calculate $H(z)$, the Hubble parameter.

Parameters

z : array_like

Input redshifts.

Returns

E : ndarray, or float if input scalar

The redshift scaling of the Hubble constant.

Notes

The return value, E , is defined such that $H(z) = H_0 E$.

`inv_efunc(z)`

Function used to calculate $\frac{1}{H_z}$.

Parameters

z : array_like

Input redshifts.

Returns

E : ndarray, or float if input scalar

The inverse redshift scaling of the Hubble constant.

Notes

The return value, E , is defined such that $H_z = H_0/E$.

Flatw0waCDM

```
class astropy.cosmology.Flatw0waCDM(H0, Om0, w0=-1.0, wa=0.0, Tcmb0=2.7250000000000001,
                                     Neff=3.04, m_nu=<Quantity 0.0 eV>, name=None)
```

Bases: `astropy.cosmology.w0waCDM`

FLRW cosmology with a CPL dark energy equation of state and no curvature.

The equation for the dark energy equation of state uses the CPL form as described in Chevallier & Polarski Int. J. Mod. Phys. D10, 213 (2001) and Linder PRL 90, 91301 (2003): $w(z) = w_0 + w_a(1 - a) = w_0 + w_a z / (1 + z)$.

Parameters

H0 : float or `Quantity`

Hubble constant at $z = 0$. If a float, must be in [km/sec/Mpc]

Om0 : float

Omega matter: density of non-relativistic matter in units of the critical density at $z=0$.

w0 : float

Dark energy equation of state at $z=0$ ($a=1$). This is pressure/density for dark energy in units where $c=1$.

wa : float

Negative derivative of the dark energy equation of state with respect to the scale factor. A cosmological constant has $w_0=-1.0$ and $w_a=0.0$.

Tcmb0 : float or `Quantity`

Temperature of the CMB $z=0$. If a float, must be in [K]. Default: 2.725.

Neff : float

Effective number of Neutrino species. Default 3.04.

m_nu : `Quantity`

Mass of each neutrino species. If this is a scalar `Quantity`, then all neutrino species are assumed to have that mass. Otherwise, the mass of each species. The actual number of neutrino species (and hence the number of elements of `m_nu` if it is not scalar) must be the floor of `Neff`. Usually this means you must provide three neutrino masses unless you are considering something like a sterile neutrino.

name : str

Optional name for this cosmological object.

Examples

```
>>> from astropy.cosmology import Flatw0waCDM
>>> cosmo = Flatw0waCDM(H0=70, Om0=0.3, w0=-0.9, wa=0.2)
```

The comoving distance in Mpc at redshift z :

```
>>> z = 0.5
>>> dc = cosmo.comoving_distance(z)
```

FlatwCDM

```
class astropy.cosmology.FlatwCDM(H0, Om0, w0=-1.0, Tcmb0=2.7250000000000001, Neff=3.04,
                                m_nu=<Quantity 0.0 eV>, name=None)
```

Bases: `astropy.cosmology.wCDM`

FLRW cosmology with a constant dark energy equation of state and no spatial curvature.

This has one additional attribute beyond those of FLRW.

Parameters

H0 : float or `Quantity`

Hubble constant at $z = 0$. If a float, must be in [km/sec/Mpc]

Om0 : float

Omega matter: density of non-relativistic matter in units of the critical density at $z=0$.

w0 : float

Dark energy equation of state at all redshifts. This is pressure/density for dark energy in units where $c=1$. A cosmological constant has $w0=-1.0$.

Tcmb0 : float or `Quantity`

Temperature of the CMB $z=0$. If a float, must be in [K]. Default: 2.725.

Neff : float

Effective number of Neutrino species. Default 3.04.

m_nu : `Quantity`

Mass of each neutrino species. If this is a scalar `Quantity`, then all neutrino species are assumed to have that mass. Otherwise, the mass of each species. The actual number of neutrino species (and hence the number of elements of `m_nu` if it is not scalar) must be the floor of `Neff`. Usually this means you must provide three neutrino masses unless you are considering something like a sterile neutrino.

name : str

Optional name for this cosmological object.

Examples

```
>>> from astropy.cosmology import FlatwCDM
>>> cosmo = FlatwCDM(H0=70, Om0=0.3, w0=-0.9)
```

The comoving distance in Mpc at redshift z :

```
>>> z = 0.5
>>> dc = cosmo.comoving_distance(z)
```

Methods Summary

<code>efunc(z)</code>	Function used to calculate $H(z)$, the Hubble parameter.
<code>inv_efunc(z)</code>	Function used to calculate $\frac{1}{H_z}$.

Methods Documentation

`efunc` (*z*)

Function used to calculate $H(z)$, the Hubble parameter.

Parameters

z : array_like

Input redshifts.

Returns

E : ndarray, or float if input scalar

The redshift scaling of the Hubble constant.

Notes

The return value, *E*, is defined such that $H(z) = H_0 E$.

`inv_efunc` (*z*)

Function used to calculate $\frac{1}{H_z}$.

Parameters

z : array_like

Input redshifts.

Returns

E : ndarray, or float if input scalar

The inverse redshift scaling of the Hubble constant.

Notes

The return value, *E*, is defined such that $H_z = H_0/E$.

LambdaCDM

```
class astropy.cosmology.LambdaCDM(H0, Om0, Ode0, Tcmb0=2.7250000000000001, Neff=3.04,  
                                m_nu=<Quantity 0.0 eV>, name=None)
```

Bases: `astropy.cosmology.FLRW`

FLRW cosmology with a cosmological constant and curvature.

This has no additional attributes beyond those of FLRW.

Parameters

H0 : float or `Quantity`

Hubble constant at $z = 0$. If a float, must be in [km/sec/Mpc]

Om0 : float

Omega matter: density of non-relativistic matter in units of the critical density at $z=0$.

Ode0 : float

Omega dark energy: density of the cosmological constant in units of the critical density at $z=0$.

Tcmb0 : float or `Quantity`

Temperature of the CMB $z=0$. If a float, must be in [K]. Default: 2.725.

Neff : float

Effective number of Neutrino species. Default 3.04.

m_nu : `Quantity`

Mass of each neutrino species. If this is a scalar `Quantity`, then all neutrino species are assumed to have that mass. Otherwise, the mass of each species. The actual number of neutrino species (and hence the number of elements of `m_nu` if it is not scalar) must be the floor of `Neff`. Usually this means you must provide three neutrino masses unless you are considering something like a sterile neutrino.

name : str

Optional name for this cosmological object.

Examples

```
>>> from astropy.cosmology import LambdaCDM
>>> cosmo = LambdaCDM(H0=70, Om0=0.3, Ode0=0.7)
```

The comoving distance in Mpc at redshift z :

```
>>> z = 0.5
>>> dc = cosmo.comoving_distance(z)
```

Methods Summary

<code>de_density_scale(z)</code>	Evaluates the redshift dependence of the dark energy density.
<code>efunc(z)</code>	Function used to calculate $H(z)$, the Hubble parameter.
<code>inv_efunc(z)</code>	Function used to calculate $\frac{1}{H_z}$.
<code>w(z)</code>	Returns dark energy equation of state at redshift z .

Methods Documentation

de_density_scale (z)

Evaluates the redshift dependence of the dark energy density.

Parameters

z : array_like

Input redshifts.

Returns

I : ndarray, or float if input scalar

The scaling of the energy density of dark energy with redshift.

Notes

The scaling factor, I , is defined by $\rho(z) = \rho_0 I$, and in this case is given by $I = 1$.

efunc (*z*)

Function used to calculate $H(z)$, the Hubble parameter.

Parameters

z : array_like

Input redshifts.

Returns

E : ndarray, or float if input scalar

The redshift scaling of the Hubble constant.

Notes

The return value, E , is defined such that $H(z) = H_0 E$.

inv_efunc (*z*)

Function used to calculate $\frac{1}{H_z}$.

Parameters

z : array_like

Input redshifts.

Returns

E : ndarray, or float if input scalar

The inverse redshift scaling of the Hubble constant.

Notes

The return value, E , is defined such that $H_z = H_0/E$.

w (*z*)

Returns dark energy equation of state at redshift z .

Parameters

z : array_like

Input redshifts.

Returns

w : ndarray, or float if input scalar

The dark energy equation of state

Notes

The dark energy equation of state is defined as $w(z) = P(z)/\rho(z)$, where $P(z)$ is the pressure at redshift z and $\rho(z)$ is the density at redshift z , both in units where $c=1$. Here this is $w(z) = -1$.

default_cosmology

class `astropy.cosmology.default_cosmology`

Bases: `astropy.utils.state.ScienceState`

The default cosmology to use. To change it:

```
>>> from astropy.cosmology import default_cosmology, WMAP7
>>> with default_cosmology.set(WMAP7):
...     # WMAP7 cosmology in effect
```

Or, you may use a string:

```
>>> with default_cosmology.set('WMAP7'):
...     # WMAP7 cosmology in effect
```

Methods Summary

<code>get_cosmology_from_string(arg)</code>	Return a cosmology instance from a string.
<code>validate(value)</code>	

Methods Documentation

static `get_cosmology_from_string` (*arg*)

Return a cosmology instance from a string.

classmethod `validate` (*value*)

w0waCDM

class `astropy.cosmology.w0waCDM` (*H0, Om0, Ode0, w0=-1.0, wa=0.0, Tcmb0=2.7250000000000001, Neff=3.04, m_nu=<Quantity 0.0 eV>, name=None*)

Bases: `astropy.cosmology.FLRW`

FLRW cosmology with a CPL dark energy equation of state and curvature.

The equation for the dark energy equation of state uses the CPL form as described in Chevallier & Polarski Int. J. Mod. Phys. D10, 213 (2001) and Linder PRL 90, 91301 (2003): $w(z) = w_0 + w_a(1-a) = w_0 + w_a z / (1+z)$.

Parameters

H0 : float or `Quantity`

Hubble constant at $z = 0$. If a float, must be in [km/sec/Mpc]

Om0 : float

Omega matter: density of non-relativistic matter in units of the critical density at $z=0$.

Ode0 : float

Omega dark energy: density of dark energy in units of the critical density at $z=0$.

w0 : float

Dark energy equation of state at $z=0$ ($a=1$). This is pressure/density for dark energy in units where $c=1$.

wa : float

Negative derivative of the dark energy equation of state with respect to the scale factor. A cosmological constant has $w0=-1.0$ and $wa=0.0$.

Tcmb0 : float or `Quantity`

Temperature of the CMB $z=0$. If a float, must be in [K]. Default: 2.725.

Neff : float

Effective number of Neutrino species. Default 3.04.

m_nu : `Quantity`

Mass of each neutrino species. If this is a scalar `Quantity`, then all neutrino species are assumed to have that mass. Otherwise, the mass of each species. The actual number of neutrino species (and hence the number of elements of `m_nu` if it is not scalar) must be the floor of `Neff`. Usually this means you must provide three neutrino masses unless you are considering something like a sterile neutrino.

name : str

Optional name for this cosmological object.

Examples

```
>>> from astropy.cosmology import w0waCDM
>>> cosmo = w0waCDM(H0=70, Om0=0.3, Ode0=0.7, w0=-0.9, wa=0.2)
```

The comoving distance in Mpc at redshift z :

```
>>> z = 0.5
>>> dc = cosmo.comoving_distance(z)
```

Attributes Summary

<code>w0</code>	Dark energy equation of state at $z=0$
<code>wa</code>	Negative derivative of dark energy equation of state w.r.t.

Methods Summary

<code>de_density_scale(z)</code>	Evaluates the redshift dependence of the dark energy density.
<code>w(z)</code>	Returns dark energy equation of state at redshift z .

Attributes Documentation

w0

Dark energy equation of state at $z=0$

wa

Negative derivative of dark energy equation of state w.r.t. a

Methods Documentation

de_density_scale (z)

Evaluates the redshift dependence of the dark energy density.

Parameters

z : `array_like`

Input redshifts.

Returns

I : ndarray, or float if input scalar

The scaling of the energy density of dark energy with redshift.

Notes

The scaling factor, I, is defined by $\rho(z) = \rho_0 I$, and in this case is given by

$$I = (1 + z)^{3(1+w_0+w_a)} \exp\left(-3w_a \frac{z}{1+z}\right)$$

w(z)

Returns dark energy equation of state at redshift z.

Parameters

z : array_like

Input redshifts.

Returns

w : ndarray, or float if input scalar

The dark energy equation of state

Notes

The dark energy equation of state is defined as $w(z) = P(z)/\rho(z)$, where $P(z)$ is the pressure at redshift z and $\rho(z)$ is the density at redshift z, both in units where c=1. Here this is $w(z) = w_0 + w_a(1 - a) = w_0 + w_a \frac{z}{1+z}$.

w0wzCDM

```
class astropy.cosmology.w0wzCDM(H0, Om0, Ode0, w0=-1.0, wz=0.0, Tcmb0=2.7250000000000001,
                                Neff=3.04, m_nu=<Quantity 0.0 eV>, name=None)
```

Bases: `astropy.cosmology.FLRW`

FLRW cosmology with a variable dark energy equation of state and curvature.

The equation for the dark energy equation of state uses the simple form: $w(z) = w_0 + w_z z$.

This form is not recommended for $z > 1$.

Parameters

H0 : float or `Quantity`

Hubble constant at z = 0. If a float, must be in [km/sec/Mpc]

Om0 : float

Omega matter: density of non-relativistic matter in units of the critical density at z=0.

Ode0 : float

Omega dark energy: density of dark energy in units of the critical density at z=0.

w0 : float

Dark energy equation of state at $z=0$. This is pressure/density for dark energy in units where $c=1$. A cosmological constant has $w_0=-1.0$.

wz : float

Derivative of the dark energy equation of state with respect to z .

Tcmb0 : float or [Quantity](#)

Temperature of the CMB $z=0$. If a float, must be in [K]. Default: 2.725.

Neff : float

Effective number of Neutrino species. Default 3.04.

m_nu : float or ndarray or [Quantity](#)

Mass of each neutrino species, in eV. If this is a float or scalar [Quantity](#), then all neutrino species are assumed to have that mass. If a ndarray or array [Quantity](#), then these are the values of the mass of each species. The actual number of neutrino species (and hence the number of elements of `m_nu` if it is not scalar) must be the floor of `Neff`. Usually this means you must provide three neutrino masses unless you are considering something like a sterile neutrino.

name : str

Optional name for this cosmological object.

Examples

```
>>> from astropy.cosmology import w0wzCDM
>>> cosmo = w0wzCDM(H0=70, Om0=0.3, Ode0=0.7, w0=-0.9, wz=0.2)
```

The comoving distance in Mpc at redshift z :

```
>>> z = 0.5
>>> dc = cosmo.comoving_distance(z)
```

Attributes Summary

<code>w0</code>	Dark energy equation of state at $z=0$
<code>wz</code>	Derivative of the dark energy equation of state w.r.t.

Methods Summary

<code>de_density_scale(z)</code>	Evaluates the redshift dependence of the dark energy density.
<code>w(z)</code>	Returns dark energy equation of state at redshift z .

Attributes Documentation

w0

Dark energy equation of state at $z=0$

wz

Derivative of the dark energy equation of state w.r.t. z

Methods Documentation

`de_density_scale` (*z*)

Evaluates the redshift dependence of the dark energy density.

Parameters

z : array_like

Input redshifts.

Returns

I : ndarray, or float if input scalar

The scaling of the energy density of dark energy with redshift.

Notes

The scaling factor, I, is defined by $\rho(z) = \rho_0 I$, and in this case is given by

$$I = (1 + z)^{3(1+w_0-w_z)} \exp(-3w_z z)$$

`w` (*z*)

Returns dark energy equation of state at redshift *z*.

Parameters

z : array_like

Input redshifts.

Returns

w : ndarray, or float if input scalar

The dark energy equation of state

Notes

The dark energy equation of state is defined as $w(z) = P(z)/\rho(z)$, where $P(z)$ is the pressure at redshift *z* and $\rho(z)$ is the density at redshift *z*, both in units where $c=1$. Here this is given by $w(z) = w_0 + w_z z$.

wCDM

```
class astropy.cosmology.wCDM(H0, Om0, Ode0, w0=-1.0, Tcmb0=2.7250000000000001, Neff=3.04,
                             m_nu=<Quantity 0.0 eV>, name=None)
```

Bases: `astropy.cosmology.FLRW`

FLRW cosmology with a constant dark energy equation of state and curvature.

This has one additional attribute beyond those of FLRW.

Parameters

H0 : float or `Quantity`

Hubble constant at $z = 0$. If a float, must be in [km/sec/Mpc]

Om0 : float

Omega matter: density of non-relativistic matter in units of the critical density at $z=0$.

Ode0 : float

Omega dark energy: density of dark energy in units of the critical density at $z=0$.

w0 : float

Dark energy equation of state at all redshifts. This is pressure/density for dark energy in units where $c=1$. A cosmological constant has $w_0=-1.0$.

Tcmb0 : float or `Quantity`

Temperature of the CMB $z=0$. If a float, must be in [K]. Default: 2.725.

Neff : float

Effective number of Neutrino species. Default 3.04.

m_nu : `Quantity`

Mass of each neutrino species. If this is a scalar `Quantity`, then all neutrino species are assumed to have that mass. Otherwise, the mass of each species. The actual number of neutrino species (and hence the number of elements of `m_nu` if it is not scalar) must be the floor of `Neff`. Usually this means you must provide three neutrino masses unless you are considering something like a sterile neutrino.

name : str

Optional name for this cosmological object.

Examples

```
>>> from astropy.cosmology import wCDM
>>> cosmo = wCDM(H0=70, Om0=0.3, Ode0=0.7, w0=-0.9)
```

The comoving distance in Mpc at redshift z :

```
>>> z = 0.5
>>> dc = cosmo.comoving_distance(z)
```

Attributes Summary

w0	Dark energy equation of state
----	-------------------------------

Methods Summary

<code>de_density_scale(z)</code>	Evaluates the redshift dependence of the dark energy density.
<code>efunc(z)</code>	Function used to calculate $H(z)$, the Hubble parameter.
<code>inv_efunc(z)</code>	Function used to calculate $\frac{1}{H_z}$.
<code>w(z)</code>	Returns dark energy equation of state at redshift z .

Attributes Documentation

w0

Dark energy equation of state

Methods Documentation**de_density_scale** (*z*)

Evaluates the redshift dependence of the dark energy density.

Parameters

z : array_like

Input redshifts.

Returns

I : ndarray, or float if input scalar

The scaling of the energy density of dark energy with redshift.

Notes

The scaling factor, I, is defined by $\rho(z) = \rho_0 I$, and in this case is given by $I = (1 + z)^{3(1+w_0)}$

efunc (*z*)

Function used to calculate $H(z)$, the Hubble parameter.

Parameters

z : array_like

Input redshifts.

Returns

E : ndarray, or float if input scalar

The redshift scaling of the Hubble constant.

Notes

The return value, E, is defined such that $H(z) = H_0 E$.

inv_efunc (*z*)

Function used to calculate $\frac{1}{H_z}$.

Parameters

z : array_like

Input redshifts.

Returns

E : ndarray, or float if input scalar

The inverse redshift scaling of the Hubble constant.

Notes

The return value, E, is defined such that $H_z = H_0/E$.

w (*z*)

Returns dark energy equation of state at redshift *z*.

Parameters

z : array_like

Input redshifts.

Returns

w : ndarray, or float if input scalar

The dark energy equation of state

Notes

The dark energy equation of state is defined as $w(z) = P(z)/\rho(z)$, where $P(z)$ is the pressure at redshift z and $\rho(z)$ is the density at redshift z , both in units where $c=1$. Here this is $w(z) = w_0$.

wpwaCDM

```
class astropy.cosmology.wpwaCDM(H0, Om0, Ode0, wp=-1.0, wa=0.0, zp=0,
                                Tcmb0=2.7250000000000001, Neff=3.04, m_nu=<Quantity
                                0.0 eV>, name=None)
```

Bases: `astropy.cosmology.FLRW`

FLRW cosmology with a CPL dark energy equation of state, a pivot redshift, and curvature.

The equation for the dark energy equation of state uses the CPL form as described in Chevallier & Polarski Int. J. Mod. Phys. D10, 213 (2001) and Linder PRL 90, 91301 (2003), but modified to have a pivot redshift as in the findings of the Dark Energy Task Force (Albrecht et al. arXiv:0901.0721 (2009)): $w(a) = w_p + w_a(a_p - a) = w_p + w_a(1/(1 + zp) - 1/(1 + z))$.

Parameters

H0 : float or `Quantity`

Hubble constant at $z = 0$. If a float, must be in [km/sec/Mpc]

Om0 : float

Omega matter: density of non-relativistic matter in units of the critical density at $z=0$.

Ode0 : float

Omega dark energy: density of dark energy in units of the critical density at $z=0$.

wp : float

Dark energy equation of state at the pivot redshift z_p . This is pressure/density for dark energy in units where $c=1$.

wa : float

Negative derivative of the dark energy equation of state with respect to the scale factor. A cosmological constant has $w_0=-1.0$ and $w_a=0.0$.

zp : float

Pivot redshift – the redshift where $w(z) = w_p$

Tcmb0 : float or `Quantity`

Temperature of the CMB $z=0$. If a float, must be in [K]. Default: 2.725.

Neff : float

Effective number of Neutrino species. Default 3.04.

m_nu : `Quantity`

Mass of each neutrino species. If this is a scalar Quantity, then all neutrino species are assumed to have that mass. Otherwise, the mass of each species. The actual number of neutrino species (and hence the number of elements of `m_nu` if it is not scalar) must be the floor of `Neff`. Usually this means you must provide three neutrino masses unless you are considering something like a sterile neutrino.

name : str

Optional name for this cosmological object.

Examples

```
>>> from astropy.cosmology import wpwaCDM
>>> cosmo = wpwaCDM(H0=70, Om0=0.3, Ode0=0.7, wp=-0.9, wa=0.2, zp=0.4)
```

The comoving distance in Mpc at redshift `z`:

```
>>> z = 0.5
>>> dc = cosmo.comoving_distance(z)
```

Attributes Summary

<code>wa</code>	Negative derivative of dark energy equation of state w.r.t.
<code>wp</code>	Dark energy equation of state at the pivot redshift <code>zp</code>
<code>zp</code>	The pivot redshift, where $w(z) = wp$

Methods Summary

<code>de_density_scale(z)</code>	Evaluates the redshift dependence of the dark energy density.
<code>w(z)</code>	Returns dark energy equation of state at redshift <code>z</code> .

Attributes Documentation

wa
Negative derivative of dark energy equation of state w.r.t. `a`

wp
Dark energy equation of state at the pivot redshift `zp`

zp
The pivot redshift, where $w(z) = wp$

Methods Documentation

de_density_scale (`z`)
Evaluates the redshift dependence of the dark energy density.

Parameters

`z` : array_like

Input redshifts.

Returns**I** : ndarray, or float if input scalar

The scaling of the energy density of dark energy with redshift.

NotesThe scaling factor, I, is defined by $\rho(z) = \rho_0 I$, and in this case is given by

$$a_p = \frac{1}{1 + z_p}$$
$$I = (1 + z)^{3(1+w_p+a_p w_a)} \exp\left(-3w_a \frac{z}{1+z}\right)$$

w(z)

Returns dark energy equation of state at redshift z.

Parameters**z** : array_like

Input redshifts.

Returns**w** : ndarray, or float if input scalar

The dark energy equation of state

Notes

The dark energy equation of state is defined as $w(z) = P(z)/\rho(z)$, where $P(z)$ is the pressure at redshift z and $\rho(z)$ is the density at redshift z, both in units where $c=1$. Here this is $w(z) = w_p + w_a(a_p - a)$ where $a = 1/1 + z$ and $a_p = 1/1 + z_p$.

Class Inheritance Diagram

ASTROSTATISTICS TOOLS (ASTROPY . STATS)

20.1 Introduction

The `astropy.stats` package holds statistical functions or algorithms used in astronomy and astropy.

20.2 Getting Started

The current tools are fairly self-contained, and include relevant examples in their docstrings.

20.3 See Also

- `scipy.stats`

This `scipy` package contains a variety of useful statistical functions and classes. The functionality in `astropy.stats` is intended to supplement this, *not* replace it.

20.4 Reference/API

20.4.1 `astropy.stats` Module

This subpackage contains statistical tools provided for or used by Astropy.

While the `scipy.stats` package contains a wide range of statistical tools, it is a general-purpose package, and is missing some that are particularly useful to astronomy or are used in an atypical way in astronomy. This package is intended to provide such functionality, but *not* to replace `scipy.stats` if its implementation satisfies astronomers' needs.

Functions

<code>binned_binom_proportion(x, success[, bins, ...])</code>	Binomial proportion and confidence interval in bins of a continuous variab
<code>binom_conf_interval(k, n[, conf, interval])</code>	Binomial proportion confidence interval given k successes, n trials.
<code>biweight_location(a[, c, M])</code>	Compute the biweight location for an array.
<code>biweight_midvariance(a[, c, M])</code>	Compute the biweight midvariance for an array.
<code>bootstrap(data[, bootnum, samples, bootfunc])</code>	Performs bootstrap resampling on numpy arrays.
<code>median_absolute_deviation(a[, axis])</code>	Compute the median absolute deviation.

Continue

Table 20.1 – continued from previous page

<code>sigma_clip(data[, sig, iters, cenfunc, ...])</code>	Perform sigma-clipping on the provided data.
<code>signal_to_noise_oir_ccd(t, source_eps, ...)</code>	Computes the signal to noise ratio for source being observed in the optical

binned_binom_proportion

`astropy.stats.binned_binom_proportion(x, success, bins=10, range=None, conf=0.6826900000000002, interval='wilson')`

Binomial proportion and confidence interval in bins of a continuous variable x .

Given a set of datapoint pairs where the x values are continuously distributed and the `success` values are binomial (“success / failure” or “true / false”), place the pairs into bins according to x value and calculate the binomial proportion (fraction of successes) and confidence interval in each bin.

Parameters

x : list_like

Values.

success : list_like (bool)

Success (`True`) or failure (`False`) corresponding to each value in x . Must be same length as x .

bins : int or sequence of scalars, optional

If `bins` is an int, it defines the number of equal-width bins in the given range (10, by default). If `bins` is a sequence, it defines the bin edges, including the rightmost edge, allowing for non-uniform bin widths (in this case, ‘range’ is ignored).

range : (float, float), optional

The lower and upper range of the bins. If `None` (default), the range is set to $(x.min(), x.max())$. Values outside the range are ignored.

conf : float in [0, 1], optional

Desired probability content in the confidence interval ($p - perr[0]$, $p + perr[1]$) in each bin. Default is 0.68269.

interval : {‘wilson’, ‘jeffreys’, ‘flat’, ‘wald’}, optional

Formula used to calculate confidence interval on the binomial proportion in each bin. See `binom_conf_interval` for definition of the intervals. The ‘wilson’, ‘jeffreys’, and ‘flat’ intervals generally give similar results. ‘wilson’ should be somewhat faster, while ‘jeffreys’ and ‘flat’ are marginally superior, but differ in the assumed prior. The ‘wald’ interval is generally not recommended. It is provided for comparison purposes. Default is ‘wilson’.

Returns

bin_ctr : numpy.ndarray

Central value of bins. Bins without any entries are not returned.

bin_halfwidth : numpy.ndarray

Half-width of each bin such that `bin_ctr - bin_halfwidth` and `bin_ctr + bin_halfwidth` give the left and right side of each bin, respectively.

p : numpy.ndarray

Efficiency in each bin.

perr : numpy.ndarray

2-d array of shape (2, len(p)) representing the upper and lower uncertainty on p in each bin.

See also:

`binom_conf_interval`

Function used to estimate confidence interval in each bin.

Examples

Suppose we wish to estimate the efficiency of a survey in detecting astronomical sources as a function of magnitude (i.e., the probability of detecting a source given its magnitude). In a realistic case, we might prepare a large number of sources with randomly selected magnitudes, inject them into simulated images, and then record which were detected at the end of the reduction pipeline. As a toy example, we generate 100 data points with randomly selected magnitudes between 20 and 30 and “observe” them with a known detection function (here, the error function, with 50% detection probability at magnitude 25):

```
>>> from scipy.special import erf
>>> from scipy.stats.distributions import binom
>>> def true_efficiency(x):
...     return 0.5 + 0.5 * erf((x - 25.) / 2.)
>>> mag = 20. + 10. * np.random.rand(100)
>>> detected = binom.rvs(1, true_efficiency(mag))
>>> bins, binshw, p, perr = binned_binom_proportion(mag, detected, bins=20)
>>> plt.errorbar(bins, p, xerr=binshw, yerr=perr, ls='none', marker='o',
...              label='estimate')
```

The above example uses the Wilson confidence interval to calculate the uncertainty `perr` in each bin (see the definition of various confidence intervals in `binom_conf_interval`). A commonly used alternative is the Wald interval. However, the Wald interval can give nonsensical uncertainties when the efficiency is near 0 or 1, and is therefore **not** recommended. As an illustration, the following example shows the same data as above but uses the Wald interval rather than the Wilson interval to calculate `perr`:

```
>>> bins, binshw, p, perr = binned_binom_proportion(mag, detected, bins=20,
...                                                interval='wald')
>>> plt.errorbar(bins, p, xerr=binshw, yerr=perr, ls='none', marker='o',
...              label='estimate')
```

`binom_conf_interval`

`astropy.stats.binom_conf_interval(k, n, conf=0.6826900000000002, interval='wilson')`

Binomial proportion confidence interval given k successes, n trials.

Parameters

k : int or numpy.ndarray

Number of successes ($0 \leq k \leq n$).

n : int or numpy.ndarray

Number of trials ($n > 0$). If both k and n are arrays, they must have the same shape.

conf : float in [0, 1], optional

Desired probability content of interval. Default is 0.68269, corresponding to 1 sigma in a 1-dimensional Gaussian distribution.

interval : {'wilson', 'jeffreys', 'flat', 'wald'}, optional

Formula used for confidence interval. See notes for details. The 'wilson' and 'jeffreys' intervals generally give similar results, while 'flat' is somewhat different, especially for small values of n. 'wilson' should be somewhat faster than 'flat' or 'jeffreys'. The 'wald' interval is generally not recommended. It is provided for comparison purposes. Default is 'wilson'.

Returns

conf_interval : numpy.ndarray

conf_interval[0] and conf_interval[1] correspond to the lower and upper limits, respectively, for each element in k, n.

Notes

In situations where a probability of success is not known, it can be estimated from a number of trials (N) and number of observed successes (k). For example, this is done in Monte Carlo experiments designed to estimate a detection efficiency. It is simple to take the sample proportion of successes (k/N) as a reasonable best estimate of the true probability ϵ . However, deriving an accurate confidence interval on ϵ is non-trivial. There are several formulas for this interval (see [R10]). Four intervals are implemented here:

1. The Wilson Interval. This interval, attributed to Wilson [R11], is given by

$$CI_{\text{Wilson}} = \frac{k + \kappa^2/2}{N + \kappa^2} \pm \frac{\kappa n^{1/2}}{n + \kappa^2} ((\hat{\epsilon}(1 - \hat{\epsilon}) + \kappa^2/(4n))^{1/2})$$

where $\hat{\epsilon} = k/N$ and κ is the number of standard deviations corresponding to the desired confidence interval for a normal distribution (for example, 1.0 for a confidence interval of 68.269%). For a confidence interval of 100(1 - α)%,

$$\kappa = \Phi^{-1}(1 - \alpha/2) = \sqrt{2}\text{erf}^{-1}(1 - \alpha).$$

2. The Jeffreys Interval. This interval is derived by applying Bayes' theorem to the binomial distribution with the noninformative Jeffreys prior [R12], [R13]. The noninformative Jeffreys prior is the Beta distribution, Beta(1/2, 1/2), which has the density function

$$f(\epsilon) = \pi^{-1} \epsilon^{-1/2} (1 - \epsilon)^{-1/2}.$$

The justification for this prior is that it is invariant under reparameterizations of the binomial proportion. The posterior density function is also a Beta distribution: Beta(k + 1/2, N - k + 1/2). The interval is then chosen so that it is *equal-tailed*: Each tail (outside the interval) contains $\alpha/2$ of the posterior probability, and the interval itself contains 1 - α . This interval must be calculated numerically. Additionally, when k = 0 the lower limit is set to 0 and when k = N the upper limit is set to 1, so that in these cases, there is only one tail containing $\alpha/2$ and the interval itself contains 1 - $\alpha/2$ rather than the nominal 1 - α .

3. A Flat prior. This is similar to the Jeffreys interval, but uses a flat (uniform) prior on the binomial proportion over the range 0 to 1 rather than the reparameterization-invariant Jeffreys prior. The posterior density function is a Beta distribution: Beta(k + 1, N - k + 1). The same comments about the nature of the interval (equal-tailed, etc.) also apply to this option.

4. The Wald Interval. This interval is given by

$$CI_{\text{Wald}} = \hat{\epsilon} \pm \kappa \sqrt{\frac{\hat{\epsilon}(1 - \hat{\epsilon})}{N}}$$

The Wald interval gives acceptable results in some limiting cases. Particularly, when N is very large, and the true proportion ϵ is not "too close" to 0 or 1. However, as the later is not verifiable when trying to estimate ϵ , this is not very helpful. Its use is not recommended, but it is provided here for comparison purposes due to its prevalence in everyday practical statistics.

References

[R10], [R11], [R12], [R13]

Examples

Integer inputs return an array with shape (2,):

```
>>> binom_conf_interval(4, 5, interval='wilson')
array([ 0.57921724,  0.92078259])
```

Arrays of arbitrary dimension are supported. The Wilson and Jeffreys intervals give similar results, even for small k, N:

```
>>> binom_conf_interval([0, 1, 2, 5], 5, interval='wilson')
array([[ 0.          ,  0.07921741,  0.21597328,  0.83333304],
       [ 0.16666696,  0.42078276,  0.61736012,  1.          ]])
```

```
>>> binom_conf_interval([0, 1, 2, 5], 5, interval='jeffreys')
array([[ 0.          ,  0.0842525 ,  0.21789949,  0.82788246],
       [ 0.17211754,  0.42218001,  0.61753691,  1.          ]])
```

```
>>> binom_conf_interval([0, 1, 2, 5], 5, interval='flat')
array([[ 0.          ,  0.12139799,  0.24309021,  0.73577037],
       [ 0.26422963,  0.45401727,  0.61535699,  1.          ]])
```

In contrast, the Wald interval gives poor results for small k, N. For k = 0 or k = N, the interval always has zero length.

```
>>> binom_conf_interval([0, 1, 2, 5], 5, interval='wald')
array([[ 0.          ,  0.02111437,  0.18091075,  1.          ],
       [ 0.          ,  0.37888563,  0.61908925,  1.          ]])
```

For confidence intervals approaching 1, the Wald interval for $0 < k < N$ can give intervals that extend outside [0, 1]:

```
>>> binom_conf_interval([0, 1, 2, 5], 5, interval='wald', conf=0.99)
array([[ 0.          , -0.26077835, -0.16433593,  1.          ],
       [ 0.          ,  0.66077835,  0.96433593,  1.          ]])
```

biweight_location

`astropy.stats.biweight_location(a, c=6.0, M=None)`

Compute the biweight location for an array.

Returns the biweight location for the array elements. The biweight is a robust statistic for determining the central location of a distribution.

The biweight location is given by the following equation

$$C_{bl} = M + \frac{\sum_{\|u_i\| < 1} (x_i - M)(1 - u_i^2)^2}{\sum_{\|u_i\| < 1} (1 - u_i^2)^2}$$

where M is the sample mean or if run iterative the initial guess, and u_i is given by

$$u_i = \frac{(x_i - M)}{cMAD}$$

where MAD is the median absolute deviation.

For more details, see Beers, Flynn, and Gebhardt, 1990, AJ, 100, 32B

Parameters

a : array_like

Input array or object that can be converted to an array.

c : float

Tuning constant for the biweight estimator. Default value is 6.0.

M : float, optional

Initial guess for the biweight location.

Returns

biweight_location: float

Returns the biweight location for the array elements.

See also:

`median_absolute_deviation`, `biweight_midvariance`

Examples

This will generate random variates from a Gaussian distribution and return the median absolute deviation for that distribution:

```
>>> from astropy.stats.funcs import biweight_location
>>> from numpy.random import randn
>>> randvar = randn(10000)
>>> cbl = biweight_location(randvar)
```

biweight_midvariance

`astropy.stats.biweight_midvariance` (*a*, *c=9.0*, *M=None*)

Compute the biweight midvariance for an array.

Returns the biweight midvariance for the array elements. The biweight midvariance is a robust statistic for determining the midvariance (i.e. the standard deviation) of a distribution.

The biweight location is given by the following equation

$$C_{bl} = n^{1/2} \frac{[\sum_{|u_i| < 1} (x_i - M) * * 2(1 - u_i^2)^4]^{0.5}}{|\sum_{|u_i| < 1} (1 - u_i^2)(1 - 5u_i^2)|}$$

where u_i is given by

$$u_i = \frac{(x_i - M)}{cMAD}$$

where MAD is the median absolute deviation. For the midvariance parameter, *c* is typically uses a value of 9.0.

For more details, see Beers, Flynn, and Gebhardt, 1990, AJ, 100, 32B

Parameters

a : array_like

Input array or object that can be converted to an array.

c : float

Tuning constant for the biweight estimator. Default value is 9.0.

M : float, optional

Initial guess for the biweight location.

Returns

biweight_midvariance : float

Returns the biweight midvariance for the array elements.

See also:

`median_absolute_deviation`, `biweight_location`

Examples

This will generate random variates from a Gaussian distribution and return the median absolute deviation for that distribution:

```
>>> from astropy.stats.funcs import biweight_midvariance
>>> from numpy.random import randn
>>> randvar = randn(10000)
>>> scl = biweight_midvariance(randvar)
```

bootstrap

`astropy.stats.bootstrap` (*data*, *bootnum=100*, *samples=None*, *bootfunc=None*)

Performs bootstrap resampling on numpy arrays.

Bootstrap resampling is used to understand confidence intervals of sample estimates. This function returns versions of the dataset resampled with replacement (“case bootstrapping”). These can all be run through a function or statistic to produce a distribution of values which can then be used to find the confidence intervals.

Parameters

data : numpy.ndarray

N-D array. The bootstrap resampling will be performed on the first index, so the first index should access the relevant information to be bootstrapped.

bootnum : int

Number of bootstrap resamples

samples : int

Number of samples in each resample. The default `None` sets samples to the number of datapoints

bootfunc : function

Function to reduce the resampled data. Each bootstrap resample will be put through this function and the results returned. If `None`, the bootstrapped data will be returned

Returns

boot : numpy.ndarray

Bootstrapped data. Each row is a bootstrap resample of the data.

median_absolute_deviation

`astropy.stats.median_absolute_deviation(a, axis=None)`

Compute the median absolute deviation.

Returns the median absolute deviation (MAD) of the array elements. The MAD is defined as $\text{median}(\text{abs}(a - \text{median}(a)))$.

Parameters

a : array_like

Input array or object that can be converted to an array.

axis : int, optional

Axis along which the medians are computed. The default (`axis=None`) is to compute the median along a flattened version of the array.

Returns

median_absolute_deviation : ndarray

A new array holding the result. If the input contains integers, or floats of smaller precision than 64, then the output data-type is float64. Otherwise, the output data-type is the same as that of the input.

See also:

[numpy.median](#)

Examples

This will generate random variates from a Gaussian distribution and return the median absolute deviation for that distribution:

```
>>> from astropy.stats import median_absolute_deviation
>>> from numpy.random import randn
>>> randvar = randn(10000)
>>> mad = median_absolute_deviation(randvar)
```

sigma_clip

`astropy.stats.sigma_clip(data, sig=3, iters=1, cenfunc=<function median at 0x264df50>, varfunc=<function var at 0x229ade8>, axis=None, copy=True)`

Perform sigma-clipping on the provided data.

This performs the sigma clipping algorithm - i.e. the data will be iterated over, each time rejecting points that are more than a specified number of standard deviations discrepant.

Note: `scipy.stats.sigmadclip` provides a subset of the functionality in this function.

Parameters

data : array-like

The data to be sigma-clipped (any shape).

sig : float

The number of standard deviations (*not* variances) to use as the clipping limit.

iters : int or `None`

The number of iterations to perform clipping for, or `None` to clip until convergence is achieved (i.e. continue until the last iteration clips nothing).

cenfunc : callable

The technique to compute the center for the clipping. Must be a callable that takes in a masked array and outputs the central value. Defaults to the median (`numpy.median`).

varfunc : callable

The technique to compute the standard deviation about the center. Must be a callable that takes in a masked array and outputs a width estimator:

```
deviation**2 > sig**2 * varfunc(deviation)
```

Defaults to the variance (`numpy.var`).

axis : int or `None`

If not `None`, clip along the given axis. For this case, `axis=int` will be passed on to `cenfunc` and `varfunc`, which are expected to return an array with the axis dimension removed (like the `numpy` functions). If `None`, clip over all values. Defaults to `None`.

copy : bool

If `True`, the data array will be copied. If `False`, the masked array data will contain the same array as `data`. Defaults to `True`.

Returns

filtered_data : `numpy.ma.MaskedArray`

A masked array with the same shape as `data` input, where the points rejected by the algorithm have been masked.

Notes

1. The routine works by calculating:

```
deviation = data - cenfunc(data [,axis=int])
```

and then setting a mask for points outside the range:

```
data.mask = deviation**2 > sig**2 * varfunc(deviation)
```

It will iterate a given number of times, or until no further points are rejected.

2. Most `numpy` functions deal well with masked arrays, but if one would like to have an array with just the good (or bad) values, one can use:

```
good_only = filtered_data.data[~filtered_data.mask]
bad_only = filtered_data.data[filtered_data.mask]
```

However, for multidimensional data, this flattens the array, which may not be what one wants (especially if filtering was done along an axis).

Examples

This will generate random variates from a Gaussian distribution and return a masked array in which all points that are more than 2 *sample* standard deviation from the median are masked:

```
>>> from astropy.stats import sigma_clip
>>> from numpy.random import randn
>>> randvar = randn(10000)
>>> filtered_data = sigma_clip(randvar, 2, 1)
```

This will clipping on a similar distribution, but for 3 sigma relative to the sample *mean*, will clip until converged, and does not copy the data:

```
>>> from astropy.stats import sigma_clip
>>> from numpy.random import randn
>>> from numpy import mean
>>> randvar = randn(10000)
>>> filtered_data = sigma_clip(randvar, 3, None, mean, copy=False)
```

This will clip along one axis on a similar distribution with bad points inserted:

```
>>> from astropy.stats import sigma_clip
>>> from numpy.random import normal
>>> from numpy import arange, diag, ones
>>> data = arange(5)+normal(0.,0.05,(5,5))+diag(ones(5))
>>> filtered_data = sigma_clip(data, axis=0, sig=2.3)
```

Note that along the other axis, no points would be masked, as the variance is higher.

signal_to_noise_oir_ccd

`astropy.stats.signal_to_noise_oir_ccd(t, source_eps, sky_eps, dark_eps, rd, npix, gain=1.0)`
Computes the signal to noise ratio for source being observed in the optical/IR using a CCD.

Parameters

t : float or numpy.ndarray

CCD integration time in seconds

source_eps : float

Number of electrons (photons) or DN per second in the aperture from the source. Note that this should already have been scaled by the filter transmission and the quantum efficiency of the CCD. If the input is in DN, then be sure to set the gain to the proper value for the CCD. If the input is in electrons per second, then keep the gain as its default of 1.0.

sky_eps : float

Number of electrons (photons) or DN per second per pixel from the sky background. Should already be scaled by filter transmission and QE. This must be in the same units as `source_eps` for the calculation to make sense.

dark_eps : float

Number of thermal electrons per second per pixel. If this is given in DN or ADU, then multiply by the gain to get the value in electrons.

rd : float

Read noise of the CCD in electrons. If this is given in DN or ADU, then multiply by the gain to get the value in electrons.

npix : float

Size of the aperture in pixels

gain : float

Gain of the CCD. In units of electrons per DN.

Returns

SNR : float or numpy.ndarray

Signal to noise ratio calculated from the inputs

VIRTUAL OBSERVATORY ACCESS (ASTROPY.VO)

21.1 Introduction

The `astropy.vo` subpackage handles simple access for Virtual Observatory (VO) services.

Current services include:

21.1.1 VO Simple Cone Search

Astropy offers Simple Cone Search Version 1.03 as defined in IVOA Recommendation (February 22, 2008). Cone Search queries an area encompassed by a given radius centered on a given RA and DEC and returns all the objects found within the area in the given catalog.

Default Cone Search Services

Currently, the default Cone Search services used are a subset of those found in the STScI VAO Registry. They were hand-picked to represent commonly used catalogs below:

- 2MASS All-Sky
- HST Guide Star Catalog
- SDSS Data Release 7
- SDSS-III Data Release 8
- USNO A1
- USNO A2
- USNO B1

This subset undergoes daily validations hosted by STScI using *Validation for Simple Cone Search*. Those that pass without critical warnings or exceptions are used by *Simple Cone Search* by default. They are controlled by `astropy.vo.Conf.conesearch_dbname`:

1. `'conesearch_good'` Default. Passed validation without critical warnings and exceptions.
2. `'conesearch_warn'` Has critical warnings but no exceptions. Use at your own risk.
3. `'conesearch_exception'` Has some exceptions. *Never* use this.
4. `'conesearch_error'` Has network connection error. *Never* use this.

If you are a Cone Search service provider and would like to include your service in the list above, please open a [GitHub issue on Astropy](#).

Caching

Caching of downloaded contents is controlled by `astropy.utils.data`. To use cached data, some functions in this package have a `cache` keyword that can be set to `True`.

Getting Started

This section only contains minimal examples showing how to perform basic Cone Search.

```
>>> from astropy.vo.client import conesearch
```

List the available Cone Search catalogs:

```
>>> conesearch.list_catalogs()
[u'Guide Star Catalog 2.3 1',
 u'SDSS DR7 - Sloan Digital Sky Survey Data Release 7 1',
 u'SDSS DR7 - Sloan Digital Sky Survey Data Release 7 2',
 u'SDSS DR7 - Sloan Digital Sky Survey Data Release 7 3',
 u'SDSS DR7 - Sloan Digital Sky Survey Data Release 7 4',
 u'SDSS DR8 - Sloan Digital Sky Survey Data Release 8 1',
 u'SDSS DR8 - Sloan Digital Sky Survey Data Release 8 2',
 u'The HST Guide Star Catalog, Version 1.1 (Lasker+ 1992) 1',
 u'The HST Guide Star Catalog, Version 1.2 (Lasker+ 1996) 1',
 u'The HST Guide Star Catalog, Version GSC-ACT (Lasker+ 1996-99) 1',
 u'The PMM USNO-A1.0 Catalogue (Monet 1997) 1',
 u'The USNO-A2.0 Catalogue (Monet+ 1998) 1',
 u'Two Micron All Sky Survey (2MASS) 1',
 u'Two Micron All Sky Survey (2MASS) 2',
 u'USNO-A2 Catalogue 1',
 u'USNO-A2.0 1']
```

Select a 2MASS catalog from the list above that is to be searched:

```
>>> my_catname = 'Two Micron All Sky Survey (2MASS) 1'
```

Query the selected 2MASS catalog around M31 with a 0.1-degree search radius:

```
>>> from astropy.coordinates import SkyCoord
>>> from astropy import units as u
>>> c = SkyCoord.from_name('M31')
>>> c.ra, c.dec
(<Longitude 10.6847083 deg>, <Latitude 41.26875 deg>)
>>> result = conesearch.conesearch(c, 0.1 * u.degree, catalog_db=my_catname)
Trying http://wfaudata.roe.ac.uk/twomass-dsa/DirectCone?DSACAT=TWOMASS&...
Downloading ...
WARNING: W06: ... UCD has invalid character '?' in '??' [...]
WARNING: W50: ... Invalid unit string 'yyyy-mm-dd' [...]
WARNING: W50: ... Invalid unit string 'Julian days' [...]
>>> result
<Table rows=2008 names=('cx', 'cy', ...>
>>> result.url
u'http://wfaudata.roe.ac.uk/twomass-dsa/DirectCone?DSACAT=TWOMASS&DSATAB=twomass_psc&'
```

Get the number of matches and returned column names:

```
>>> result.array.size
2008
>>> result.array.dtype.names
('cx',
```

```
'cy',
'cz',
'htmID',
'ra',
'dec', ...,
'coadd_key',
'coadd')
```

Extract RA and DEC of the matches:

```
>>> result.array['ra']
masked_array(data = [10.620983 10.672264 10.651166 ..., 10.805599],
             mask = [False False False ..., False],
             fill_value = 1e+20)
>>> result.array['dec']
masked_array(data = [41.192303 41.19426 41.19445 ..., 41.262123],
             mask = [False False False ..., False],
             fill_value = 1e+20)
```

Using `astropy.vo`

This package has four main components across two subpackages:

Using `astropy.vo.client`

This subpackage contains modules supporting VO client-side operations.

Catalog Manipulation You can manipulate a VO catalog using `VOSCatalog`, which is basically a dictionary with added functionalities.

Examples

```
>>> from astropy.vo.client.vos_catalog import VOSCatalog
```

You can create a VO catalog from scratch with your own VO service by providing its title and access URL, and optionally any other metadata as key-value pairs:

```
>>> my_cat = VOSCatalog.create(
...     'My Own', 'http://ex.org/cgi-bin/cs.pl?',
...     description='My first VO service.', creator='J. Doe', year=2013)
>>> print(my_cat)
title: My Own
url: http://ex.org/cgi-bin/cs.pl?
>>> print(my_cat.dumps())
{
  "creator": "J. Doe",
  "description": "My first VO service.",
  "title": "My Own",
  "url": "http://ex.org/cgi-bin/cs.pl?",
  "year": 2013
}
```

You can modify and add fields:

```
>>> my_cat['year'] = 2014
>>> my_cat['new_field'] = 'Hello world'
>>> print(my_cat.dumps())
{
  "creator": "J. Doe",
  "description": "My first VO service.",
  "new_field": "Hello world",
  "title": "My Own",
  "url": "http://ex.org/cgi-bin/cs.pl?",
  "year": 2014
}
```

In addition, you can also delete an existing field, except the compulsory title and access URL:

```
>>> my_cat.delete_attribute('description')
>>> print(my_cat.dumps())
{
  "creator": "J. Doe",
  "new_field": "Hello world",
  "title": "My Own",
  "url": "http://ex.org/cgi-bin/cs.pl?",
  "year": 2014
}
```

Database Manipulation You can manipulate VO database using `VOSDatabase`, which is basically a nested dictionary with added functionalities.

Examples

```
>>> from astropy.vo.client.vos_catalog import VOSDatabase
```

You can choose to start with an empty database:

```
>>> my_db = VOSDatabase.create_empty()
>>> print(my_db.dumps())
{
  "__version__": 1,
  "catalogs": {}
}
```

Add the custom catalog from *VO catalog examples* to database:

```
>>> my_db.add_catalog('My Catalog 1', my_cat)
>>> print(my_db)
My Catalog 1
>>> print(my_db.dumps())
{
  "__version__": 1,
  "catalogs": {
    "My Catalog 1": {
      "creator": "J. Doe",
      "new_field": "Hello world",
      "title": "My Own",
      "url": "http://ex.org/cgi-bin/cs.pl?",
      "year": 2014
    }
  }
}
```

You can write/read the new database to/from a JSON file:

```
>>> my_db.to_json('my_vo_database.json', clobber=True)
>>> my_db = VODatabase.from_json('my_vo_database.json')
```

You can also load a database from a VO registry. The process is described in *Building the Database from Registry*, except that here, validation is not done, so `validate_xxx` keys are not added. This might generate a lot of warnings, especially if the registry has duplicate entries of similar services, so here, we silently ignore all the warnings:

```
>>> import warnings
>>> from astropy.vo.validator.validate import CS_MSTR_LIST
>>> with warnings.catch_warnings():
...     warnings.simplefilter('ignore')
...     registry_db = VODatabase.from_registry(
...         CS_MSTR_LIST(), encoding='binary', cache=False)
Downloading http://vao.stsci.edu/directory/NVORegInt.asmx/...
|=====| 25M/ 25M (100.00%) 00s
>>> len(registry_db)
11937
```

Find catalog names containing 'usno*a2' in the registry database:

```
>>> usno_a2_list = registry_db.list_catalogs(pattern='usno*a2')
>>> usno_a2_list
[u'ROSAT All-Sky Survey Bright Source Catalog USNO A2 Cross-Associations 1',
 u'The USNO-A2.0 Catalogue (Monet+ 1998) 1',
 u'USNO-A2 Catalogue 1',
 u'USNO-A2.0 1',
 u'USNO-SA2.0 1']
```

Find access URLs containing 'stsci' in the registry database:

```
>>> stsci_urls = registry_db.list_catalogs_by_url(pattern='stsci')
>>> stsci_urls
['http://archive.stsci.edu/befs/search.php?',
 'http://archive.stsci.edu/copernicus/search.php?', ...,
 'http://galex.stsci.edu/gxWS/ConeSearch/gxConeSearch.aspx?',
 'http://gsss.stsci.edu/webservices/vo/ConeSearch.aspx?CAT=GSC23&']
```

Extract a catalog titled 'USNO-A2 Catalogue 1' from the registry:

```
>>> usno_a2 = registry_db.get_catalog('USNO-A2 Catalogue 1')
>>> print(usno_a2)
title: USNO-A2 Catalogue
url: http://www.nofs.navy.mil/cgi-bin/vo_cone.cgi?CAT=USNO-A2&
```

Extract a catalog by known access URL from the registry (the iterator version of this functionality is `get_catalogs_by_url()`, which is useful in the case of multiple entries with same access URL):

```
>>> gsc = registry_db.get_catalog_by_url(
...     'http://vizier.u-strasbg.fr/viz-bin/votable/-A?-source=I/305/out&')
>>> print(gsc)
title: The Guide Star Catalog, Version 2.3.2 (GSC2.3) (STScI, 2006)
url: http://vizier.u-strasbg.fr/viz-bin/votable/-A?-source=I/305/out&
```

Add all 'usno*a2' catalogs from registry to your database:

```
>>> for name, cat in registry_db.get_catalogs():
...     if name in usno_a2_list:
...         my_db.add_catalog(name, cat)
```

```
>>> my_db.list_catalogs()
[u'My Catalog 1',
 u'ROSAT All-Sky Survey Bright Source Catalog USNO A2 Cross-Associations 1',
 u'The USNO-A2.0 Catalogue (Monet+ 1998) 1',
 u'USNO-A2 Catalogue 1',
 u'USNO-A2.0 1',
 u'USNO-SA2.0 1']
```

You can delete a catalog from the database either by name or access URL:

```
>>> my_db.delete_catalog('USNO-SA2.0 1')
>>> my_db.delete_catalog_by_url(
...     'http://www.nofs.navy.mil/cgi-bin/vo_cone.cgi?CAT=USNO-A2&')
>>> my_db.list_catalogs()
[u'My Catalog 1',
 u'ROSAT All-Sky Survey Bright Source Catalog USNO A2 Cross-Associations 1',
 u'The USNO-A2.0 Catalogue (Monet+ 1998) 1',
 u'USNO-A2.0 1']
```

You can also merge two database together. In this example, the second database contains a simple catalog that only has given name and access URL:

```
>>> other_db = VOSDatabase.create_empty()
>>> other_db.add_catalog_by_url(
...     'My Guide Star Catalogue',
...     'http://vizier.u-strasbg.fr/viz-bin/votable/-A?-source=I/305/out&')
>>> print(other_db.dumps())
{
  "__version__": 1,
  "catalogs": {
    "My Guide Star Catalogue": {
      "title": "My Guide Star Catalogue",
      "url": "http://vizier.u-strasbg.fr/viz-bin/votable/-A?-source=I/305/out&"
    }
  }
}
>>> merged_db = my_db.merge(other_db)
>>> merged_db.list_catalogs()
[u'My Catalog 1',
 u'My Guide Star Catalogue',
 u'ROSAT All-Sky Survey Bright Source Catalog USNO A2 Cross-Associations 1',
 u'The USNO-A2.0 Catalogue (Monet+ 1998) 1',
 u'USNO-A2.0 1']
```

General VO Services Access `astropy.vo.client.vos_catalog` also contains common utilities for accessing simple VO services already validated by STScI (see *Validation for Simple Cone Search*).

Configurable Items These parameters are set via *Configuration system* (`astropy.config`):

- `astropy.io.votable.Conf.pedantic` Set strictness of VO table parser (`False` is recommended).
- `astropy.utils.data.Conf.remote_timeout` Timeout for remote service access.
- `astropy.vo.Conf.vos_baseurl` URL (or path) where VO Service database is stored.

Examples

```
>>> from astropy.vo.client import vos_catalog
```

Get all catalogs from a database named 'conesearch_good' (this contains cone search services that cleanly passed daily validations; also see *Cone Search Examples*):

```
>>> my_db = vos_catalog.get_remote_catalog_db('conesearch_good')
Downloading http://stsdas.stsci.edu/astrolib/vo_databases/conesearch_good.json
|=====| 56/ 56k (100.00%) 00s
>>> print(my_db)
Guide Star Catalog 2.3 1
SDSS DR7 - Sloan Digital Sky Survey Data Release 7 1
SDSS DR7 - Sloan Digital Sky Survey Data Release 7 2
# ...
USNO-A2 Catalogue 1
USNO-A2.0 1
```

If you get timeout error, you need to use a custom timeout as follows:

```
>>> from astropy.utils import data
>>> with data.conf.set_temp('remote_timeout', 30):
...     my_db = vos_catalog.get_remote_catalog_db('conesearch_good')
```

To see validation warnings generated by *Validation for Simple Cone Search* for the one of the catalogs above:

```
>>> my_cat = my_db.get_catalog('Guide Star Catalog 2.3 1')
>>> for w in my_cat['validate_warnings']:
...     print(w)
/.../vo.xml:136:0: W50: Invalid unit string 'pixel'
/.../vo.xml:155:0: W48: Unknown attribute 'nrows' on TABLEDATA
```

By default, pedantic is False:

```
>>> from astropy.io.votable import conf
>>> conf.pedantic
False
```

To call a given VO service; In this case, a Cone Search (also see *Cone Search Examples*):

```
>>> from astropy import coordinates as coord
>>> from astropy import units as u
>>> c = coord.SkyCoord.from_name('47 Tuc')
>>> c
<SkyCoord (ICRS): ra=6.0223292 deg, dec=-72.0814444 deg>
>>> sr = 0.5 * u.degree
>>> sr
<Quantity 0.5 deg>
>>> result = vos_catalog.call_vo_service(
...     'conesearch_good',
...     kwargs={'RA': c.ra.degree, 'DEC': c.dec.degree, 'SR': sr.value},
...     catalog_db='The PMM USNO-A1.0 Catalogue (Monet 1997) 1')
Trying http://vizier.u-strasbg.fr/viz-bin/votable/-A?-source=I/243/out&
Downloading ...
WARNING: W22: ... The DEFINITIONS element is deprecated in VOTable 1.1...
WARNING: W03: ... Implicitly generating an ID from a name 'RA(ICRS)'...
WARNING: W03: ... Implicitly generating an ID from a name 'DE(ICRS)'...
>>> result
<Table rows=36184 names=('_r', '_RAJ2000', '_DEJ2000', ...)>
```

To repeat the above and suppress *all* the screen outputs (not recommended):

```
>>> import warnings
>>> with warnings.catch_warnings():
...     warnings.simplefilter('ignore')
...     result = vos_catalog.call_vo_service(
...         'conesearch_good',
...         kwargs={'RA': c.ra.degree, 'DEC': c.dec.degree, 'SR': sr.value},
...         catalog_db='The PMM USNO-A1.0 Catalogue (Monet 1997) 1',
...         verbose=False)
```

You can also use custom VO database, say, 'my_vo_database.json' from *VO database examples*:

```
>>> import os
>>> from astropy.vo.client.vos_catalog import BASEURL
>>> with BASEURL.set_temp(os.curdir):
...     try:
...         result = vos_catalog.call_vo_service(
...             'my_vo_database',
...             kwargs={'RA': c.ra.degree, 'DEC': c.dec.degree,
...                     'SR': sr.value})
...     except Exception as e:
...         print(e)
Trying http://ex.org/cgi-bin/cs.pl?
Downloading http://ex.org/cgi-bin/cs.pl?SR=0.5&DEC=-72.0814444&RA=6.0223292
|=====| 1.8k/1.8k (100.00%)          00s
None of the available catalogs returned valid results.
```

Simple Cone Search `astropy.vo.client.conesearch` supports VO Simple Cone Search capabilities.

Available databases are generated on the server-side hosted by STScI using *Validation for Simple Cone Search*. The database used is controlled by `astropy.vo.Conf.conesearch_dbname`, which can be changed in *Configurable Items* below. Here are the available options:

1. **'conesearch_good'**
Default. Passed validation without critical warnings and exceptions.
2. **'conesearch_warn'**
Has critical warnings but no exceptions. Use at your own risk.
3. **'conesearch_exception'**
Has some exceptions. *Never* use this.
4. **'conesearch_error'**
Has network connection error. *Never* use this.

In the default setting, it searches the good Cone Search services one by one, stops at the first one that gives non-zero match(es), and returns the result. Since the list of services are extracted from a Python dictionary, the search order might differ from call to call.

There are also functions, both synchronously and asynchronously, available to return *all* the Cone Search query results. However, this is not recommended unless one knows what one is getting into, as it could potentially take up significant run time and computing resources.

Examples below show how to use non-default search behaviors, where the user has more control of which catalog(s) to search, et cetera.

Note: Most services currently fail to parse when `pedantic=True`.

Warning: When Cone Search returns warnings, you should decide whether the results are reliable by inspecting the warning codes in `astropy.io.votable.exceptions`.

Configurable Items These parameters are set via *Configuration system (astropy.config)*:

- `astropy.vo.Conf.conesearch_dbname` Cone Search database name to query.

Also depends on *General VO Services Access Configurable Items*.

Examples

```
>>> from astropy.vo.client import conesearch
```

Shows a sorted list of Cone Search services to be searched:

```
>>> conesearch.list_catalogs()
[u'Guide Star Catalog 2.3 1',
 u'SDSS DR7 - Sloan Digital Sky Survey Data Release 7 1',
 u'SDSS DR7 - Sloan Digital Sky Survey Data Release 7 2',
 u'SDSS DR7 - Sloan Digital Sky Survey Data Release 7 3',
 u'SDSS DR7 - Sloan Digital Sky Survey Data Release 7 4',
 u'SDSS DR8 - Sloan Digital Sky Survey Data Release 8 1',
 u'SDSS DR8 - Sloan Digital Sky Survey Data Release 8 2',
 u'The HST Guide Star Catalog, Version 1.1 (Lasker+ 1992) 1',
 u'The HST Guide Star Catalog, Version 1.2 (Lasker+ 1996) 1',
 u'The HST Guide Star Catalog, Version GSC-ACT (Lasker+ 1996-99) 1',
 u'The PMM USNO-A1.0 Catalogue (Monet 1997) 1',
 u'The USNO-A2.0 Catalogue (Monet+ 1998) 1',
 u'Two Micron All Sky Survey (2MASS) 1',
 u'Two Micron All Sky Survey (2MASS) 2',
 u'USNO-A2 Catalogue 1',
 u'USNO-A2.0 1']
```

To inspect them in detail, do the following and then refer to the examples in *Database Manipulation*:

```
>>> from astropy.vo.client import vos_catalog
>>> good_db = vos_catalog.get_remote_catalog_db('conesearch_good')
```

Select a catalog to search:

```
>>> my_catname = 'The PMM USNO-A1.0 Catalogue (Monet 1997) 1'
```

By default, pedantic is False:

```
>>> from astropy.io.votable import conf
>>> conf.pedantic
False
```

Perform Cone Search in the selected catalog above for 0.5 degree radius around 47 Tucanae with minimum verbosity, if supported. The `catalog_db` keyword gives control over which catalog(s) to use. If running this for the first time, a copy of the catalogs database will be downloaded to local cache. To run this again without using cached data, set `cache=False`:

```
>>> from astropy import coordinates as coord
>>> from astropy import units as u
>>> c = coord.SkyCoord.from_name('47 Tuc')
>>> c
<SkyCoord (ICRS): ra=6.0223292 deg, dec=-72.0814444 deg>
```

```
>>> sr = 0.5 * u.degree
>>> sr
<Quantity 0.5 deg>
>>> result = conesearch.conesearch(c, sr, catalog_db=my_catname)
Trying http://vizier.u-strasbg.fr/viz-bin/votable/-A?-source=I/243/out&
Downloading ...
WARNING: W22: ... The DEFINITIONS element is deprecated in VOTable 1.1...
```

To run the command above using custom timeout of 30 seconds for each Cone Search service query:

```
>>> from astropy.utils import data
>>> with data.conf.set_temp('remote_timeout', 30):
...     result = conesearch.conesearch(c, sr, catalog_db=my_catname)
```

To suppress *all* the screen outputs (not recommended):

```
>>> import warnings
>>> with warnings.catch_warnings():
...     warnings.simplefilter('ignore')
...     result = conesearch.conesearch(c, sr, catalog_db=my_catname,
...                                     verbose=False)
```

Extract Numpy array containing the matched objects. See [numpy](#) for available operations:

```
>>> cone_arr = result.array.data
>>> cone_arr
array([(0.499298, 4.403473, -72.124045, '0150-00088188'),
       (0.499075, 4.403906, -72.122762, '0150-00088198'),
       (0.499528, 4.404531, -72.045198, '0150-00088210'), ...,
       (0.4988, 7.641731, -72.113156, '0150-00225965'),
       (0.499554, 7.645489, -72.103167, '0150-00226134'),
       (0.499917, 7.6474, -72.0876, '0150-00226223')],
      dtype=[('_r', '<f8'), ('_RAJ2000', '<f8'), ('_DEJ2000', '<f8'),
             ('USNO-A1.0', '|S13')])
>>> cone_arr.dtype.names
('_r', '_RAJ2000', '_DEJ2000', 'USNO-A1.0')
>>> cone_arr.size
36184
>>> ra_list = cone_arr['_RAJ2000']
>>> ra_list
array([ 4.403473,  4.403906,  4.404531, ...,  7.641731,  7.645489,  7.6474  ])
>>> cone_arr[0] # First row
(0.499298, 4.403473, -72.124045, '0150-00088188')
>>> cone_arr[-1] # Last row
(0.499917, 7.6474, -72.0876, '0150-00226223')
>>> cone_arr[:10] # First 10 rows
array([(0.499298, 4.403473, -72.124045, '0150-00088188'),
       (0.499075, 4.403906, -72.122762, '0150-00088198'),
       (0.499528, 4.404531, -72.045198, '0150-00088210'),
       (0.497252, 4.406078, -72.095045, '0150-00088245'),
       (0.499739, 4.406462, -72.139545, '0150-00088254'),
       (0.496312, 4.410623, -72.110492, '0150-00088372'),
       (0.49473, 4.415053, -72.071217, '0150-00088494'),
       (0.494171, 4.415939, -72.087512, '0150-00088517'),
       (0.493722, 4.417678, -72.0972, '0150-00088572'),
       (0.495147, 4.418262, -72.047142, '0150-00088595')],
      dtype=[('_r', '<f8'), ('_RAJ2000', '<f8'), ('_DEJ2000', '<f8'),
             ('USNO-A1.0', '|S13')])
```

Sort the matched objects by angular separation in ascending order:

```
>>> import numpy as np
>>> sep = cone_arr['_r']
>>> i_sorted = np.argsort(sep)
>>> cone_arr[i_sorted]
array([(0.081971, 5.917787, -72.006075, '0150-00145335'),
       (0.083181, 6.020339, -72.164623, '0150-00149799'),
       (0.089166, 5.732798, -72.077698, '0150-00137181'), ...,
       (0.499981, 7.024962, -72.477503, '0150-00198745'),
       (0.499987, 6.423773, -71.597364, '0150-00168596'),
       (0.499989, 6.899589, -72.5043, '0150-00192872')],
      dtype=[('_r', '<f8'), ('_RAJ2000', '<f8'), ('_DEJ2000', '<f8'),
            ('USNO-A1.0', '|S13')])
```

Result can also be manipulated as *VOTable XML handling* ([astropy.io.votable](http://astropy.io/votable)) and its unit can be manipulated as *Units and Quantities* ([astropy.units](http://astropy.io/units)). In this example, we convert RA values from degree to arcsec:

```
>>> from astropy import units as u
>>> ra_field = result.get_field_by_id('_RAJ2000')
>>> ra_field.title
u'Right ascension (FK5, Equinox=J2000.0) (computed by Vizier, ...)'
>>> ra_field.unit
Unit("deg")
>>> ra_field.unit.to(u.arcsec) * ra_list
array([ 15852.5028, 15854.0616, 15856.3116, ..., 27510.2316,
        27523.7604, 27530.64  ])
```

Perform the same Cone Search as above but asynchronously using `AsyncConeSearch`. Queries to individual Cone Search services are still governed by `astropy.utils.data.Conf.remote_timeout`. Cone Search is forced to run in silent mode asynchronously, but warnings are still controlled by `warnings`:

```
>>> async_search = conesearch.AsyncConeSearch(c, sr, catalog_db=my_catname)
```

Check asynchronous search status:

```
>>> async_search.running()
True
>>> async_search.done()
False
```

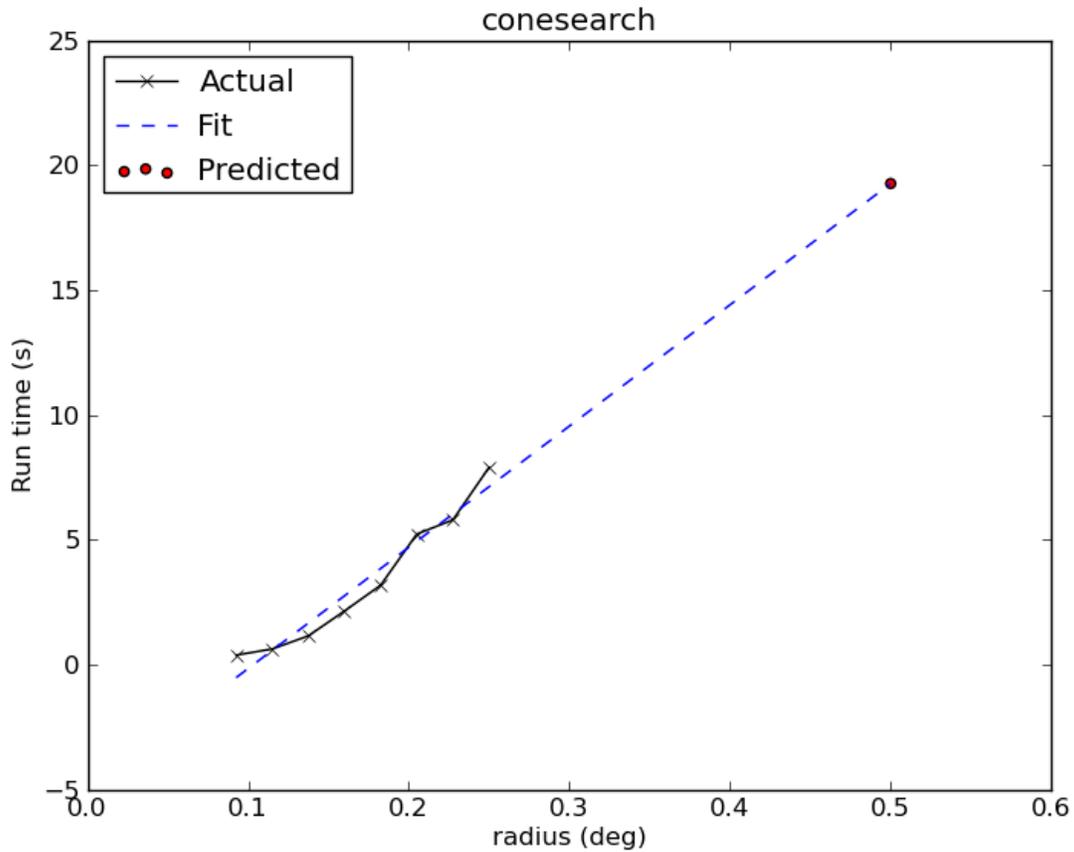
Get search results after a 30-second wait (not to be confused with `astropy.utils.data.Conf.remote_timeout` that governs individual Cone Search queries). If search is still not done after 30 seconds, `TimeoutError` is raised. Otherwise, Cone Search result is returned and can be manipulated as above. If no `timeout` keyword given, it waits until completion:

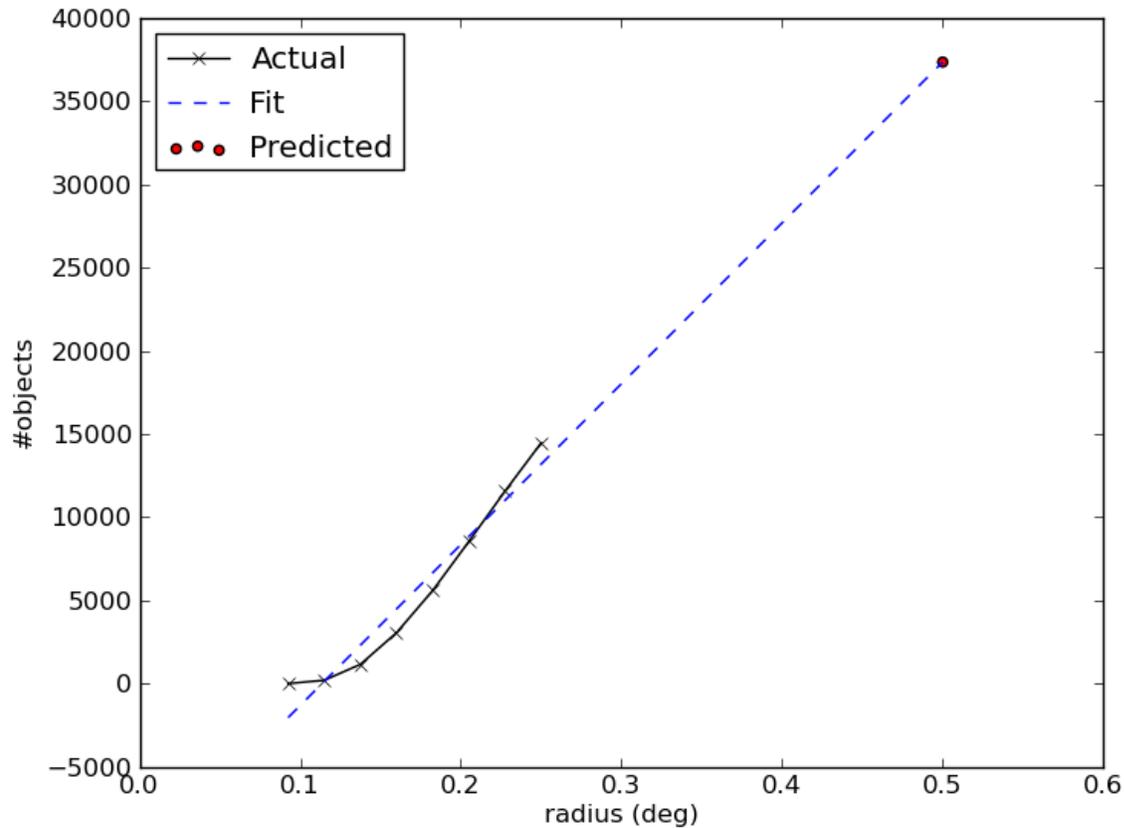
```
>>> async_result = async_search.get(timeout=30)
>>> cone_arr = async_result.array.data
>>> cone_arr.size
36184
```

Estimate the execution time and the number of objects for the Cone Search service URL from above. The prediction naively assumes a linear model, which might not be accurate for some cases. It also uses the normal `conesearch()`, not the asynchronous version. This example uses a custom timeout of 30 seconds and runs silently (except for warnings):

```
>>> result.url
u'http://vizier.u-strasbg.fr/viz-bin/votable/-A?-source=I/243/out&'
>>> with data.conf.set_temp('remote_timeout', 30):
...     t_est, n_est = conesearch.predict_search(
```

```
...         result.url, c, sr, verbose=False, plot=True)
WARNING: W22: ... The DEFINITIONS element is deprecated in VOTable 1.1...
# ...
>>> t_est # Predicted execution time
10.757875269998323
>>> n_est # Predicted number of objects
37340
```





For debugging purpose, one can obtain the actual execution time and number of objects, and compare them with the predicted values above. The INFO message shown in controlled by `astropy.logger`. Keep in mind that running this for every prediction would defeat the purpose of the prediction itself:

```
>>> t_real, tab = conesearch.conesearch_timer(
...     c, sr, catalog_db=result.url, verbose=False)
INFO: conesearch_timer took 11.5103080273 s on AVERAGE for 1 call(s). [...]
>>> t_real # Actual execution time
9.33926796913147
>>> tab.array.size # Actual number of objects
36184
```

One can also search in a list of catalogs instead of a single one. In this example, we look for all catalogs containing 'guide*star' in their titles and only perform Cone Search using those services. The first catalog in the list to successfully return non-zero result is used. Therefore, the order of catalog names given in `catalog_db` is important:

```
>>> gsc_cats = conesearch.list_catalogs(pattern='guide*star')
>>> gsc_cats
[u'Guide Star Catalog 2.3 1',
 u'The HST Guide Star Catalog, Version 1.1 (Lasker+ 1992) 1',
 u'The HST Guide Star Catalog, Version 1.2 (Lasker+ 1996) 1',
 u'The HST Guide Star Catalog, Version GSC-ACT (Lasker+ 1996-99) 1']
>>> gsc_result = conesearch.conesearch(c, sr, catalog_db=gsc_cats)
Trying http://gsss.stsci.edu/webservices/vo/ConeSearch.aspx?CAT=GSC23&
WARNING: W50: ... Invalid unit string 'pixel' [...]
WARNING: W48: ... Unknown attribute 'nrows' on TABLEDATA [...]
```

```
>>> gsc_result.array.size
74276
>>> gsc_result.url
u'http://gsss.stsci.edu/webservices/vo/ConeSearch.aspx?CAT=GSC23&'
```

To repeat the Cone Search above with the services listed in a different order:

```
>>> gsc_cats_reordered = [gsc_cats[i] for i in (3, 1, 2, 0)]
>>> gsc_cats_reordered
[u'The HST Guide Star Catalog, Version GSC-ACT (Lasker+ 1996-99) 1',
 u'The HST Guide Star Catalog, Version 1.1 (Lasker+ 1992) 1',
 u'The HST Guide Star Catalog, Version 1.2 (Lasker+ 1996) 1',
 u'Guide Star Catalog 2.3 1']
>>> gsc_result = conesearch.conesearch(c, sr, catalog_db=gsc_cats_reordered)
Trying http://vizier.u-strasbg.fr/viz-bin/votable/-A?-source=I/255/out&
Downloading ...
WARNING: W22: ... The DEFINITIONS element is deprecated in VOTable 1.1...
>>> gsc_result.array.size
2997
>>> gsc_result.url
u'http://vizier.u-strasbg.fr/viz-bin/votable/-A?-source=I/255/out&'
```

To obtain results from *all* the services above:

```
>>> all_gsc_results = conesearch.search_all(c, sr, catalog_db=gsc_cats)
Trying http://gsss.stsci.edu/webservices/vo/ConeSearch.aspx?CAT=GSC23&
Downloading ...
Trying http://vizier.u-strasbg.fr/viz-bin/votable/-A?-source=I/220/out&
Downloading ...
WARNING: W22: ... The DEFINITIONS element is deprecated in VOTable 1.1...
Trying http://vizier.u-strasbg.fr/viz-bin/votable/-A?-source=I/254/out&
Downloading ...
WARNING: W22: ... The DEFINITIONS element is deprecated in VOTable 1.1...
Trying http://vizier.u-strasbg.fr/viz-bin/votable/-A?-source=I/255/out&
Downloading ...
>>> len(all_gsc_results)
4
>>> for url, tab in all_gsc_results.items():
...     print('{0} has {1} results'.format(url, tab.array.size))
http://vizier.u-strasbg.fr/viz-bin/votable/-A?-source=I/254/out& has 2998 results
http://vizier.u-strasbg.fr/viz-bin/votable/-A?-source=I/255/out& has 2997 results
http://gsss.stsci.edu/webservices/vo/ConeSearch.aspx?CAT=GSC23& has 74276 results
http://vizier.u-strasbg.fr/viz-bin/votable/-A?-source=I/220/out& has 2997 results
```

To repeat the above asynchronously:

```
>>> async_search_all = conesearch.AsyncSearchAll(c, sr, catalog_db=gsc_cats)
>>> async_search_all.running()
True
>>> async_search_all.done()
False
>>> all_gsc_results = async_search_all.get()
```

If one is unable to obtain any results using the default Cone Search database, 'conesearch_good', that only contains sites that cleanly passed validation, one can use *Configuration system (astropy.conf)* to use another database, 'conesearch_warn', containing sites with validation warnings. One should use these sites with caution:

```
>>> from astropy.vo import conf
>>> conf.conesearch_dbname = 'conesearch_warn'
```

```

>>> conesearch.list_catalogs()
Downloading http://stsdas.stsci.edu/astrolib/vo_databases/conesearch_warn.json
|=====| 87k/ 87k (100.00%) 00s
[u'2MASS All-Sky Catalog of Point Sources (Cutri+ 2003) 1',
 u'2MASS All-Sky Point Source Catalog 1',
 u'Data release 7 of Sloan Digital Sky Survey catalogs 1',
 u'Data release 7 of Sloan Digital Sky Survey catalogs 2',
 u'Data release 7 of Sloan Digital Sky Survey catalogs 3',
 u'Data release 7 of Sloan Digital Sky Survey catalogs 4',
 u'Data release 7 of Sloan Digital Sky Survey catalogs 5',
 u'Data release 7 of Sloan Digital Sky Survey catalogs 6',
 u'The 2MASS All-Sky Catalog 1',
 u'The 2MASS All-Sky Catalog 2',
 u'The USNO-B1.0 Catalog (Monet+ 2003) 1',
 u'The USNO-B1.0 Catalog 1',
 u'USNO-A V2.0, A Catalog of Astrometric Standards 1',
 u'USNO-B1 Catalogue 1']
>>> result = conesearch.conesearch(c, sr)
Trying http://vizier.u-strasbg.fr/viz-bin/votable/-A?-source=I/284/out&
Downloading ...
WARNING: W22: ... The DEFINITIONS element is deprecated in VOTable 1.1...
>>> result.array.data.size
50000

```

You can also use custom Cone Search database, say, 'my_vo_database.json' from *VO database examples*:

```

>>> import os
>>> from astropy.vo.client.vos_catalog import BASEURL
>>> BASEURL.set(os.curdir)
>>> conesearch.CONESearch_DBNAME.set('my_vo_database')
>>> conesearch.list_catalogs()
[u'My Catalog 1']
>>> result = conesearch.conesearch(c, sr)
Trying http://ex.org/cgi-bin/cs.pl?
Downloading ...
|=====| 1.8k/1.8k (100.00%) 00s
# ...
VOSError: None of the available catalogs returned valid results.

```

Using astropy.vo.validator

VO services validator is used by STScI to support *Simple Cone Search*. Currently, only Cone Search services are supported. A typical user should not need the validator. However, this could be used by VO service providers to validate their services. Currently, any service to be validated has to be registered in STScI VAO Registry.

Validation for Simple Cone Search `astropy.vo.validator.validate` validates VO services. Currently, only Cone Search validation is done using `check_conesearch_sites()`, which utilizes underlying `astropy.io.votable.validator` library.

A master list of all available Cone Search services is obtained from `astropy.vo.validator.Conf.conesearch_master_list`, which is a URL query to STScI VAO Registry by default. However, by default, only the ones in `astropy.vo.validator.Conf.conesearch_urls` are validated (also see *Default Cone Search Services*), while the rest are skipped. There are also options to validate a user-defined list of services or all of them.

All Cone Search queries are done using RA, DEC, and SR given by `<testQuery>` XML tag in the registry, and

maximum verbosity. In an uncommon case where `<testQuery>` is not defined for a service, it uses a default search for `RA=0&DEC=0&SR=0.1`.

The results are separated into 4 groups below. Each group is stored as a JSON file of `VOSDatabase`:

1. **`conesearch_good.json`** Passed validation without critical warnings and exceptions. This database residing in `astropy.vo.Conf.vos_baseurl` is the one used by *Simple Cone Search* by default.
2. **`conesearch_warn.json`** Has critical warnings but no exceptions. Users can manually set `astropy.vo.Conf.conesearch_dbname` to use this at their own risk.
3. **`conesearch_exception.json`**
Has some exceptions. *Never* use this. For informational purpose only.
4. **`conesearch_error.json`**
Has network connection error. *Never* use this. For informational purpose only.

HTML pages summarizing the validation results are stored in `'results'` sub-directory, which also contains downloaded XML files from individual Cone Search queries.

Warnings and Exceptions A subset of `astropy.io.votable.exceptions` that is considered non-critical is defined by `astropy.vo.validator.Conf.noncritical_warnings`, which will not be flagged as bad by the validator. However, this does not change the behavior of `astropy.io.votable.Conf.pedantic`, which still needs to be set to `False` for them not to be thrown out by `conesearch()`. Despite being listed as non-critical, user is responsible to check whether the results are reliable; They should not be used blindly.

Some `units recognized by VizieR` are considered invalid by Cone Search standards. As a result, they will give the warning `'W50'`, which is non-critical by default.

User can also modify `astropy.vo.validator.Conf.noncritical_warnings` to include or exclude any warnings or exceptions, as desired. However, this should be done with caution. Adding exceptions to non-critical list is not recommended.

Building the Database from Registry Each Cone Search service is a `VOSCatalog` in a `VOSDatabase` (see *Catalog Manipulation* and *Database Manipulation*).

In the master registry, there are duplicate catalog titles with different access URLs, duplicate access URLs with different titles, duplicate catalogs with slightly different descriptions, etc.

A Cone Search service is really defined by its access URL regardless of title, description, etc. By default, `from_registry()` ensures each access URL is unique across the database. However, for user-friendly catalog listing, its title will be the catalog key, not the access URL.

In the case of two different access URLs sharing the same title, each URL will have its own database entry, with a sequence number appended to their titles (e.g., `'Title 1'` and `'Title 2'`). For consistency, even if the title does not repeat, it will still be renamed to `'Title 1'`.

In the case of the same access URL appearing multiple times in the registry, the validator will store the first catalog with that access URL and throw out the rest. However, it will keep count of the number of duplicates thrown out in the `'duplicatesIgnored'` dictionary key of the catalog kept in the database.

All the existing catalog tags will be copied over as dictionary keys, except `'accessURL'` that is renamed to `'url'` for simplicity. In addition, new keys from validation are added:

- **`validate_expected`**
Expected validation result category, e.g., `"good"`.

- **validate_network_error**
Indication for connection error.
- **validate_nexceptions**
Number of exceptions found.
- **validate_nwarnings**
Number of warnings found.
- **validate_out_db_name**
Cone Search database name this entry belongs to.
- **validate_version**
Version of validation software.
- **validate_warning_types**
List of warning codes.
- **validate_warnings**
Descriptions of the warnings.
- **validate_xmllint**
Indication of whether xmllint passed.
- **validate_xmllint_content**
Output from xmllint.

Configurable Items These parameters are set via *Configuration system (astropy.config)*:

- **astropy.vo.validator.Conf.conesearch_master_list**
VO registry query URL that should return a VO table with all the desired VO services.
- **astropy.vo.validator.Conf.conesearch_urls**
Subset of Cone Search access URLs to validate.
- **astropy.vo.validator.Conf.noncritical_warnings**
List of VO table parser warning codes that are considered non-critical.

Also depends on properties in *Simple Cone Search Configurable Items*.

Examples

```
>>> from astropy.vo.validator import validate
```

Validate default Cone Search sites with multiprocessing and write results in the current directory. Reading the master registry can be slow, so the default timeout is internally set to 60 seconds for it. However, `astropy.utils.data.REMOTE_TIMEOUT` should still be set to account for accessing the individual services (at least 30 seconds is recommended). In addition, all VO table warnings from the registry are suppressed because we are not trying to validate the registry itself but the services it contains:

```
>>> from astropy.utils import data
>>> with data.conf.set_temp('remote_timeout', 30):
...     validate.check_conesearch_sites()
Downloading http://vao.stsci.edu/directory/NVORegInt.asmx/...
|=====| 25M/ 25M (100.00%) 00s
INFO: Only 30/11938 site(s) are validated [astropy.vo.validator.validate]
# ...
INFO: good: 14 catalog(s) [astropy.vo.validator.validate]
INFO: warn: 12 catalog(s) [astropy.vo.validator.validate]
INFO: excp: 0 catalog(s) [astropy.vo.validator.validate]
INFO: nerr: 4 catalog(s) [astropy.vo.validator.validate]
```

```
INFO: total: 30 out of 30 catalog(s) [astropy.vo.validator.validate]
INFO: check_conesearch_sites took 451.05685997 s on AVERAGE...
```

Validate only Cone Search access URLs hosted by 'stsci.edu' without verbose outputs (except warnings that are controlled by `warnings`) or multiprocessing, and write results in 'subset' sub-directory instead of the current directory. For this example, we use `registry_db` from *VO database examples*:

```
>>> urls = registry_db.list_catalogs_by_url(pattern='stsci.edu')
>>> urls
['http://archive.stsci.edu/befs/search.php?',
 'http://archive.stsci.edu/copernicus/search.php?', ...,
 'http://galex.stsci.edu/gxWS/ConeSearch/gxConeSearch.aspx?',
 'http://gsss.stsci.edu/webservices/vo/ConeSearch.aspx?CAT=GSC23&']
>>> with data.conf.set_temp('remote_timeout', 30):
...     validate.check_conesearch_sites(
...         destdir='./subset', verbose=False, parallel=False, url_list=urls)
INFO: check_conesearch_sites took 84.7241549492 s on AVERAGE...
```

Add 'W24' from `astropy.io.votable.exceptions` to the list of non-critical warnings to be ignored and re-run default validation. This is *not* recommended unless you know exactly what you are doing:

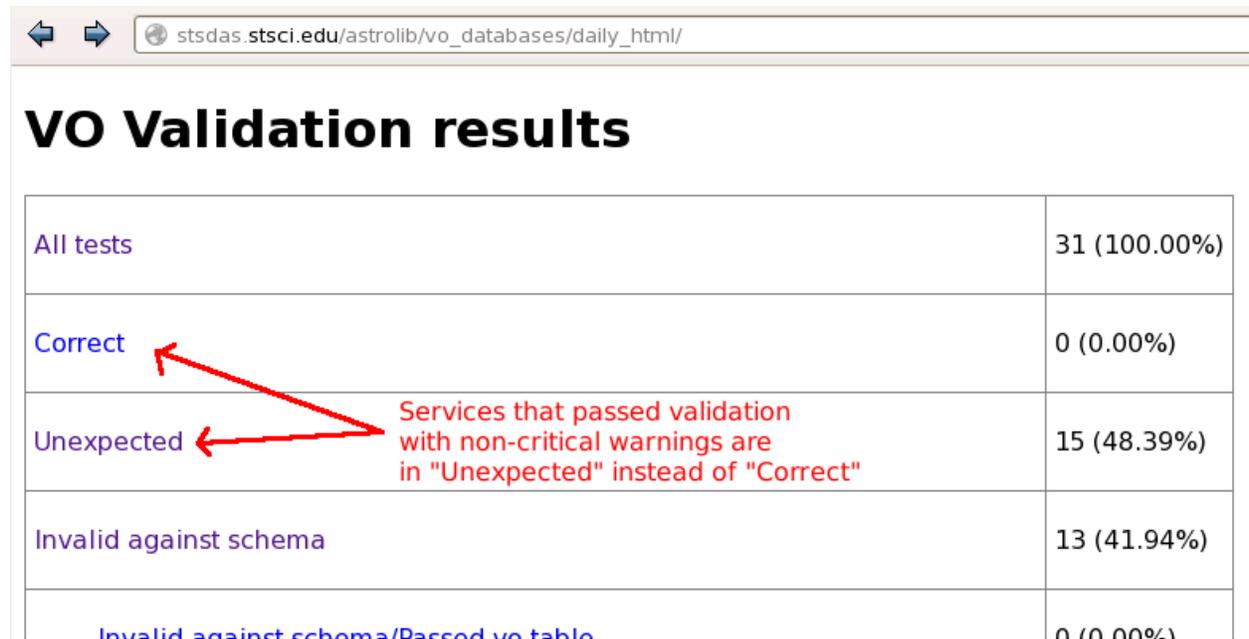
```
>>> from astropy.vo.validator.validate import conf
>>> with conf.set_temp('noncritical_warnings', conf.noncritical_warnings + ['W24']):
...     with data.conf.set_temp('remote_timeout', 30):
...         validate.check_conesearch_sites()
```

Validate *all* Cone Search services in the master registry (this will take a while) and write results in 'all' sub-directory:

```
>>> with data.conf.set_temp('remote_timeout', 30):
...     validate.check_conesearch_sites(destdir='./all', url_list=None)
```

To look at the HTML pages of the validation results in the current directory using Firefox browser (images shown are from STScI server but your own results should look similar):

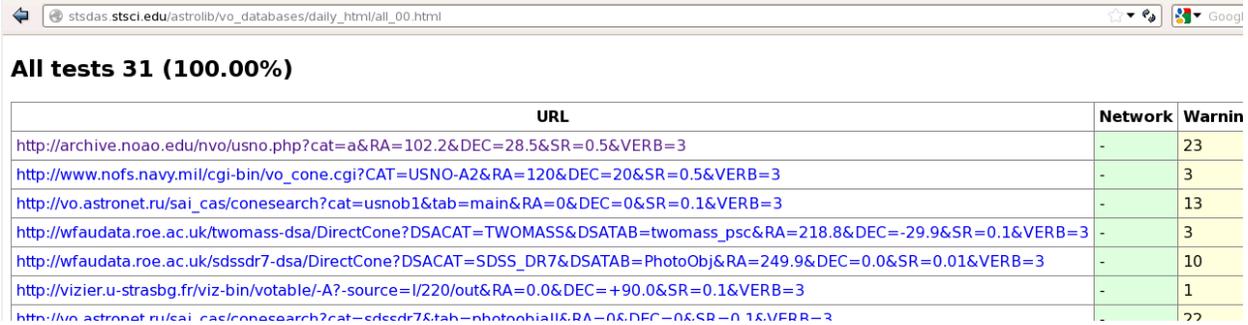
```
firefox results/index.html
```



Category	Count (Percentage)
All tests	31 (100.00%)
Correct	0 (0.00%)
Unexpected	15 (48.39%)
Invalid against schema	13 (41.94%)
Invalid against schema/Passed vo table	0 (0.00%)

Services that passed validation with non-critical warnings are in "Unexpected" instead of "Correct"

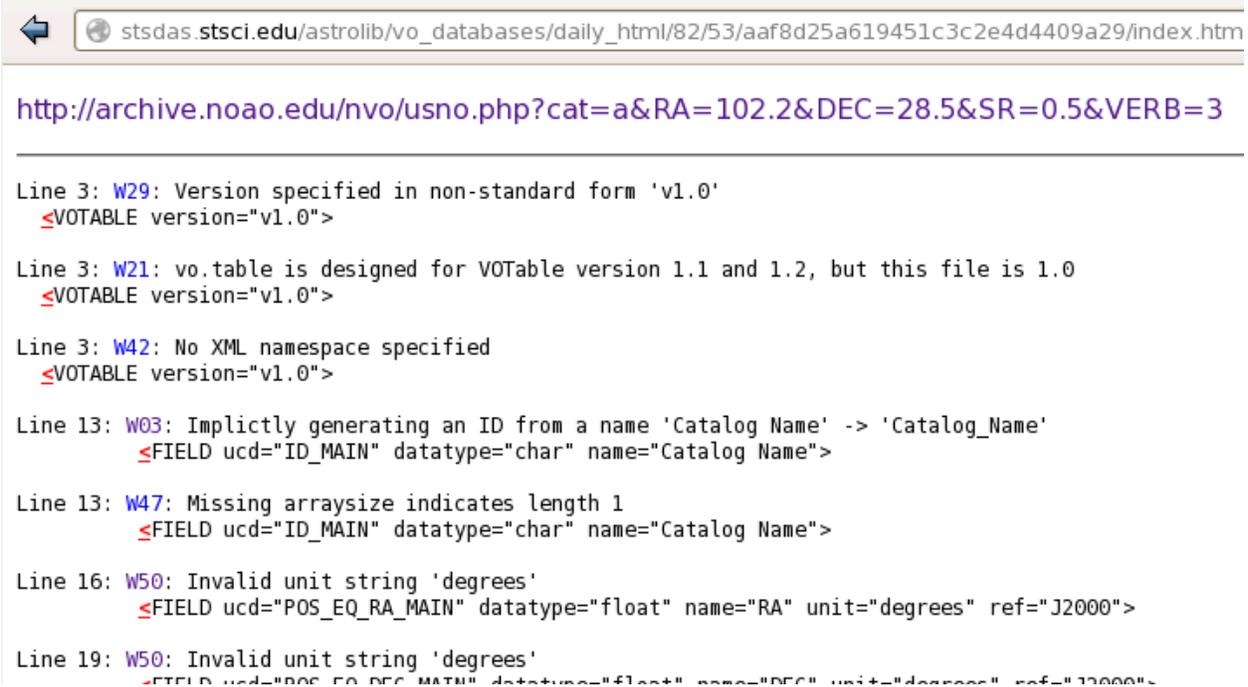
When you click on ‘All tests’ from the page above, you will see all the Cone Search services validated with a summary of validation results:



The screenshot shows a web browser window with the URL `stsdas.stsci.edu/astrolib/vo_databases/daily_html/all_00.html`. The page title is "All tests 31 (100.00%)". Below the title is a table with three columns: "URL", "Network", and "Warnin". The table lists several Cone Search services with their respective URLs, network status (indicated by a minus sign), and the number of warnings.

URL	Network	Warnin
http://archive.noao.edu/nvo/usno.php?cat=a&RA=102.2&DEC=28.5&SR=0.5&VERB=3	-	23
http://www.nofs.navy.mil/cgi-bin/vo_cone.cgi?CAT=USNO-A2&RA=120&DEC=20&SR=0.5&VERB=3	-	3
http://vo.astronet.ru/sai_cas/conesearch?cat=usnob1&tab=main&RA=0&DEC=0&SR=0.1&VERB=3	-	13
http://wfaudata.roe.ac.uk/twomass-dsa/DirectCone?DSACAT=TWOMASS&DSATAB=twomass_psc&RA=218.8&DEC=-29.9&SR=0.1&VERB=3	-	3
http://wfaudata.roe.ac.uk/sdssdr7-dsa/DirectCone?DSACAT=SDSS_DR7&DSATAB=PhotoObj&RA=249.9&DEC=0.0&SR=0.01&VERB=3	-	10
http://vizier.u-strasbg.fr/viz-bin/votable/-A?-source=I/220/out&RA=0.0&DEC=+90.0&SR=0.1&VERB=3	-	1
http://vo.astronet.ru/sai_cas/conesearch?cat=sdssdr7&tab=photoobj&RA=0&DEC=0&SR=0.1&VERB=3	-	??

When you click on any of the listed URLs from above, you will see detailed validation warnings and exceptions for the selected URL:



The screenshot shows a web browser window with the URL `stsdas.stsci.edu/astrolib/vo_databases/daily_html/82/53/aaf8d25a619451c3c2e4d4409a29/index.htm`. The page title is `http://archive.noao.edu/nvo/usno.php?cat=a&RA=102.2&DEC=28.5&SR=0.5&VERB=3`. Below the title, several validation warnings are listed, each with a line number and a warning code (W29, W21, W42, W03, W47, W50).

```

Line 3: W29: Version specified in non-standard form 'v1.0'
<VOTABLE version="v1.0">

Line 3: W21: vo.table is designed for VOTable version 1.1 and 1.2, but this file is 1.0
<VOTABLE version="v1.0">

Line 3: W42: No XML namespace specified
<VOTABLE version="v1.0">

Line 13: W03: Implicitly generating an ID from a name 'Catalog Name' -> 'Catalog_Name'
<FIELD ucd="ID_MAIN" datatype="char" name="Catalog Name">

Line 13: W47: Missing arraysize indicates length 1
<FIELD ucd="ID_MAIN" datatype="char" name="Catalog Name">

Line 16: W50: Invalid unit string 'degrees'
<FIELD ucd="POS_EQ_RA_MAIN" datatype="float" name="RA" unit="degrees" ref="J2000">

Line 19: W50: Invalid unit string 'degrees'
<FIELD ucd="POS_EQ_DEC_MAIN" datatype="float" name="DEC" unit="degrees" ref="J2000">

```

When you click on the URL on top of the page above, you will see the actual VO Table returned by the Cone Search query:

```

← stsdas.stsci.edu/astrolib/vo_databases/daily_html/82/53/aaf8d25a619451c3c2e4d4409a29/vo.xml
This XML file does not appear to have any style information associated with it. The document tree is shown
--<VOTABLE version="v1.0">
  <DESCRIPTION>NOAO USNO-A2.0 Cone Search Response</DESCRIPTION>
  -<DEFINITIONS>
    <COOSYS ID="J2000" equinox="2000.0" epoch="2000.0" system="ICRS"/>
  </DEFINITIONS>
  -<RESOURCE>
    -<TABLE>
      -<DESCRIPTION>
        USNO Catalog objects w/in 0.50 arcmin of ra=102.200000 dec=28.500000
      </DESCRIPTION>
      -<FIELD ucd="ID_MAIN" datatype="char" name="Catalog Name">
        <DESCRIPTION>USNO Object Identifier</DESCRIPTION>
      </FIELD>
      -<FIELD ucd="POS_EQ_RA_MAIN" datatype="float" name="RA" unit="degrees" ref="J2000">
        <DESCRIPTION>Right Ascension of Object (J2000)</DESCRIPTION>
      </FIELD>
      -<FIELD ucd="POS_EQ_DEC_MAIN" datatype="float" name="DEC" unit="degrees" ref="J2000">
        <DESCRIPTION>Declination of Object (J2000)</DESCRIPTION>
      </FIELD>
    </TABLE>
  </RESOURCE>
</VOTABLE>

```

Inspection of Validation Results `astropy.vo.validator.inspect` inspects results from *Validation for Simple Cone Search*. It reads in JSON files of `VOSDatabase` residing in `astropy.vo.Conf.vos_baseurl`, which can be changed to point to a different location.

Configurable Items This parameter is set via *Configuration system (astropy.config)*:

- `astropy.vo.Conf.vos_baseurl`

Examples

```
>>> from astropy.vo.validator import inspect
```

Load Cone Search validation results from `astropy.vo.Conf.vos_baseurl` (by default, the one used by *Simple Cone Search*):

```

>>> r = inspect.ConeSearchResults()
Downloading http://.../conesearch_good.json
|=====| 48k/ 48k (100.00%) 00s
Downloading http://.../conesearch_warn.json
|=====| 85k/ 85k (100.00%) 00s
Downloading http://.../conesearch_exception.json
|=====| 3.0k/3.0k (100.00%) 00s
Downloading http://.../conesearch_error.json
|=====| 4.0k/4.0k (100.00%) 00s

```

Print tally. In this example, there are 13 Cone Search services that passed validation with non-critical warnings, 14 with critical warnings, 1 with exceptions, and 2 with network error:

```

>>> r.tally()
good: 13 catalog(s)
warn: 14 catalog(s)
exception: 1 catalog(s)

```

```
error: 2 catalog(s)
total: 30 catalog(s)
```

Print a list of good Cone Search catalogs, each with title, access URL, warning codes collected, and individual warnings:

```
>>> r.list_cats('good')
Guide Star Catalog 2.3 1
http://gsss.stsci.edu/webservices/vo/ConeSearch.aspx?CAT=GSC23&
W48,W50
.../vo.xml:136:0: W50: Invalid unit string 'pixel'
.../vo.xml:155:0: W48: Unknown attribute 'nrows' on TABLEDATA
# ...
USNO-A2 Catalogue 1
http://www.nofs.navy.mil/cgi-bin/vo_cone.cgi?CAT=USNO-A2&
W17,W42,W21
.../vo.xml:4:0: W21: vo.table is designed for VOTable version 1.1 and 1.2...
.../vo.xml:4:0: W42: No XML namespace specified
.../vo.xml:15:15: W17: VOTABLE element contains more than one DESCRIPTION...
```

List Cone Search catalogs with warnings, excluding warnings that were ignored in `astropy.vo.validator.Conf.noncritical_warnings`, and writes the output to a file named `'warn_cats.txt'` in the current directory. This is useful to see why the services failed validations:

```
>>> with open('warn_cats.txt', 'w') as fout:
...     r.list_cats('warn', fout=fout, ignore_noncrit=True)
```

List the titles of all good Cone Search catalogs:

```
>>> r.catkeys['good']
[u'Guide Star Catalog 2.3 1',
 u'SDSS DR7 - Sloan Digital Sky Survey Data Release 7 1',
 u'SDSS DR7 - Sloan Digital Sky Survey Data Release 7 2',
 u'SDSS DR7 - Sloan Digital Sky Survey Data Release 7 3', ...,
 u'USNO-A2 Catalogue 1']
```

Print the details of catalog titled `'USNO-A2 Catalogue 1'`:

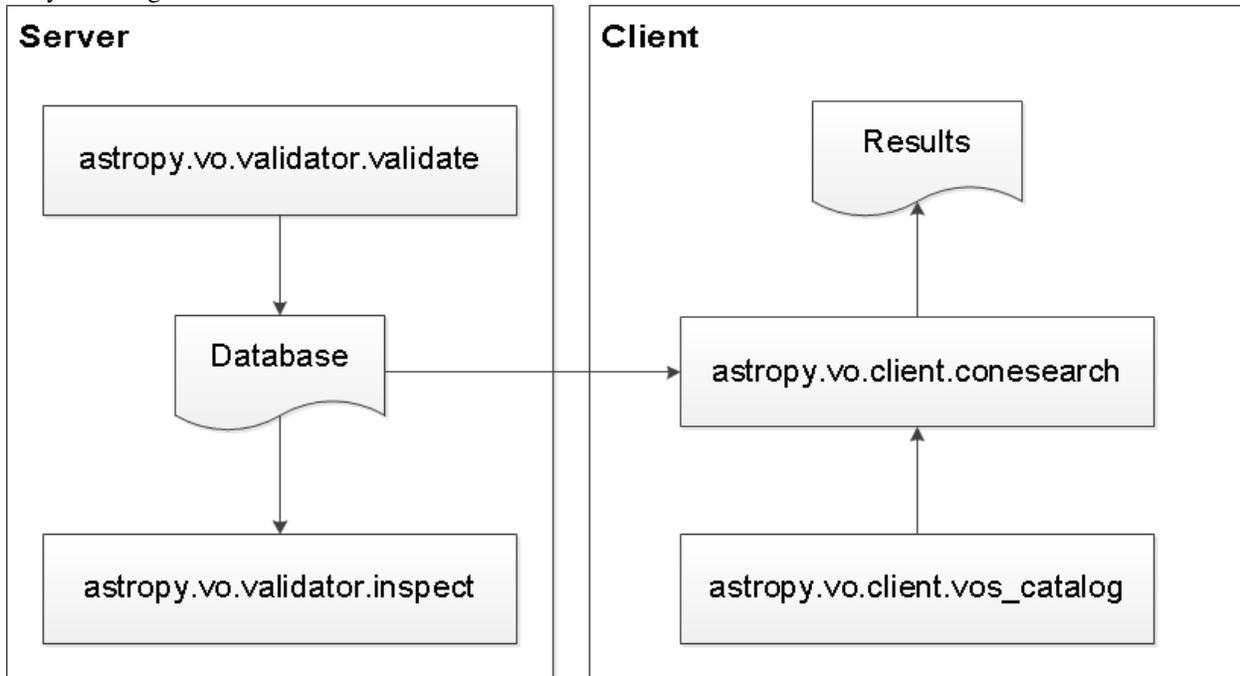
```
>>> r.print_cat('USNO-A2 Catalogue 1')
{
  "capabilityClass": "ConeSearch",
  "capabilityStandardID": "ivo://ivoa.net/std/ConeSearch",
  "capabilityValidationLevel": "",
  "contentLevel": "#University#Research#Amateur#",
  # ...
  "version": "",
  "waveband": "#Optical#"
}
Found in good
```

Load Cone Search validation results from a local directory named `'subset'`. This is useful if you ran your own *Validation for Simple Cone Search* and wish to inspect the output databases. This example reads in validation of STScI Cone Search services done in *Validation for Simple Cone Search Examples*:

```
>>> from astropy.vo import conf
>>> with conf.set_temp('vos_baseurl', './subset/'):
>>>     r = inspect.ConeSearchResults()
>>> r.tally()
good: 19 catalog(s)
```

```
warn: 7 catalog(s)
exception: 2 catalog(s)
error: 0 catalog(s)
total: 28 catalog(s)
>>> r.catkeys['good']
[u'Advanced Camera for Surveys 1',
 u'Berkeley Extreme and Far-UV Spectrometer 1',
 u'Copernicus Satellite 1',
 u'Extreme Ultraviolet Explorer 1', ...,
 u'Wisconsin Ultraviolet Photo-Polarimeter Experiment 1']
```

They are designed to be used in a work flow as illustrated below:



The one that a typical user needs is the *Simple Cone Search* component (see *Cone Search Examples*).

See Also

- NVO Directory
- Simple Cone Search Version 1.03, IVOA Recommendation (22 February 2008)
- STScI VAO Registry
- STScI VO Databases

Reference/API

astropy.vo Module

The `vo` subpackage provides virtual observatory (VO) related functionality.

Classes

Conf Configuration parameters for `astropy.vo`.

Conf**class** `astropy.vo.Conf`Bases: `astropy.config.ConfigNamespace`Configuration parameters for `astropy.vo`.**Attributes Summary**

<code>conesearch_dbname</code>	Conesearch database name to use.
<code>vos_baseurl</code>	URL where the VO Service database file is stored.

Attributes Documentation**conesearch_dbname**

Conesearch database name to use.

vos_baseurl

URL where the VO Service database file is stored.

astropy.vo.client.vos_catalog Module

Common utilities for accessing VO simple services.

Functions

<code>get_remote_catalog_db(dbname[, cache, verbose])</code>	Get a database of VO services (which is a JSON file) from a remote location.
<code>call_vo_service(service_type[, catalog_db, ...])</code>	Makes a generic VO service call.
<code>list_catalogs(service_type[, cache, verbose])</code>	List the catalogs available for the given service type.

get_remote_catalog_db
`astropy.vo.client.vos_catalog.get_remote_catalog_db(dbname, cache=True, verbose=True)`

Get a database of VO services (which is a JSON file) from a remote location.

Parameters**dbname** : strPrefix of JSON file to download from `astropy.vo.Conf.vos_baseurl`.**cache** : bool

Use caching for VO Service database. Access to actual VO websites referenced by the database still needs internet connection.

verbose : bool

Show download progress bars.

Returns**db** : `VOSDatabase`

A database of VO services.

call_vo_service

`astropy.vo.client.vos_catalog.call_vo_service` (*service_type*, *catalog_db=None*, *pedantic=None*, *verbose=True*, *cache=True*, *kwargs={}*)

Makes a generic VO service call.

Parameters

service_type : str

Name of the type of service, e.g., 'conesearch_good'. Used in error messages and to select a catalog database if `catalog_db` is not provided.

catalog_db

May be one of the following, in order from easiest to use to most control:

- None**: A database of `service_type` catalogs is downloaded from `astropy.vo.Config.vos_baseurl`. The first catalog in the database to successfully return a result is used.
- catalog name**: A name in the database of `service_type` catalogs at `astropy.vo.Config.vos_baseurl` is used. For a list of acceptable names, use `list_catalogs()`.
- url**: The prefix of a URL to a IVOA Service for `service_type`. Must end in either '?' or '&'.
- VOSCatalog** object: A specific catalog manually downloaded and selected from the database (see *General VO Services Access*).
- Any of the above 3 options combined in a list, in which case they are tried in order.

pedantic : bool or `None`

When `True`, raise an error when the file violates the spec, otherwise issue a warning. Warnings may be controlled using `warnings` module. When not provided, uses the configuration setting `astropy.io.votable.Config.pedantic`, which defaults to `False`.

verbose : bool

Verbose output.

cache : bool

Use caching for VO Service database. Access to actual VO websites referenced by the database still needs internet connection.

kwargs : dictionary

Keyword arguments to pass to the catalog service. No checking is done that the arguments are accepted by the service, etc.

Returns

obj : `astropy.io.votable.tree.Table`

First table from first successful VO service request.

Raises

VOSError

If VO service request fails.

list_catalogs

`astropy.vo.client.vos_catalog.list_catalogs` (*service_type*, *cache=True*, *verbose=True*, ***kwargs*)

List the catalogs available for the given service type.

Parameters

service_type : str

Name of the type of service, e.g., 'conesearch_good'.

cache : bool

Use caching for VO Service database. Access to actual VO websites referenced by the database still needs internet connection.

verbose : bool

Show download progress bars.

pattern : str or `None`

If given string is anywhere in a catalog name, it is considered a matching catalog. It accepts patterns as in `fnmatch` and is case-insensitive. By default, all catalogs are returned.

sort : bool

Sort output in alphabetical order. If not sorted, the order depends on dictionary hashing. Default is `True`.

Returns

arr : list of str

List of catalog names.

Classes

<code>VOSBase(tree)</code>	Base class for <code>VOSCatalog</code> and <code>VOSDatabase</code> .
<code>VOSCatalog(tree)</code>	A class to represent VO Service Catalog.
<code>VOSDatabase(tree)</code>	A class to represent a collection of <code>VOSCatalog</code> .

VOSBase

class `astropy.vo.client.vos_catalog.VOSBase` (*tree*)

Bases: `object`

Base class for `VOSCatalog` and `VOSDatabase`.

Parameters

tree : JSON tree

Methods Summary

<code>dumps()</code>	Dump the contents into a string.
----------------------	----------------------------------

Methods Documentation

`dumps ()`

Dump the contents into a string.

Returns

`s` : str

Contents as JSON string dump.

VOSCatalog

class `astropy.vo.client.vos_catalog.VOSCatalog (tree)`

Bases: `astropy.vo.client.vos_catalog.VOSBase`

A class to represent VO Service Catalog.

Parameters

tree : JSON tree

Raises

VOSError

Missing necessary key(s).

Methods Summary

<code>create(title, url, **kwargs)</code>	Create a new VO Service Catalog with user parameters.
<code>delete_attribute(key)</code>	Delete given metadata key and its value from the catalog.

Methods Documentation

classmethod `create (title, url, **kwargs)`

Create a new VO Service Catalog with user parameters.

Parameters

title : str

Title of the catalog.

url : str

Access URL of the service. This is used to build queries.

kwargs : dict

Additional metadata as keyword-value pairs describing the catalog, except 'title' and 'url'.

Returns

cat : `VOSCatalog`

VO Service Catalog.

Raises

TypeError

Multiple values given for keyword argument.

delete_attribute (*key*)

Delete given metadata key and its value from the catalog.

Parameters

key : str

Metadata key to delete.

Raises

KeyError

Key not found.

VOSError

Key must exist in catalog, therefore cannot be deleted.

VOSDatabase

class `astropy.vo.client.vos_catalog.VOSDatabase` (*tree*)

Bases: `astropy.vo.client.vos_catalog.VOSBase`

A class to represent a collection of `VOSCatalog`.

Parameters

tree : JSON tree

Raises

VOSError

If given `tree` does not have ‘catalogs’ key or catalog is invalid.

Attributes Summary

<code>version</code>	Database version number.
----------------------	--------------------------

Methods Summary

<code>add_catalog(name, cat[, allow_duplicate_url])</code>	Add a catalog to database.
<code>add_catalog_by_url(name, url, **kwargs)</code>	Like <code>add_catalog()</code> but the catalog is created with only the given name and
<code>create_empty()</code>	Create an empty database of VO services.
<code>delete_catalog(name)</code>	Delete a catalog from database with given name.
<code>delete_catalog_by_url(url)</code>	Like <code>delete_catalog()</code> but using access URL.
<code>from_json(filename, **kwargs)</code>	Create a database of VO services from a JSON file.
<code>from_registry(registry_url[, timeout])</code>	Create a database of VO services from VO registry URL.
<code>get_catalog(name)</code>	Get one catalog of given name.
<code>get_catalog_by_url(url)</code>	Like <code>get_catalog()</code> but using access URL look-up.
<code>get_catalogs()</code>	Iterator to get all catalogs.
<code>get_catalogs_by_url(url)</code>	Like <code>get_catalogs()</code> but using access URL look-up.
<code>list_catalogs([pattern, sort])</code>	List catalog names.
<code>list_catalogs_by_url([pattern, sort])</code>	Like <code>list_catalogs()</code> but using access URL.
<code>merge(other, **kwargs)</code>	Merge two database together.
<code>to_json(filename[, clobber])</code>	Write database content to a JSON file.

Attributes Documentation

version

Database version number.

Methods Documentation

add_catalog(*name*, *cat*, *allow_duplicate_url=False*)

Add a catalog to database.

Parameters

name : str

Primary key for the catalog.

cat : `VOSCatalog`

Catalog to add.

allow_duplicate_url : bool

Allow catalog with duplicate access URL?

Raises

`VOSError`

Invalid catalog.

`DuplicateCatalogName`

Catalog with given name already exists.

`DuplicateCatalogURL`

Catalog with given access URL already exists.

add_catalog_by_url(*name*, *url*, ***kwargs*)

Like `add_catalog()` but the catalog is created with only the given name and access URL.

Parameters

name : str

Primary key for the catalog.

url : str

Access URL of the service. This is used to build queries.

kwargs : dict

Keywords accepted by `add_catalog()`.

classmethod `create_empty()`

Create an empty database of VO services.

Empty database format:

```
{
  "__version__": 1,
  "catalogs" : {
  }
}
```

Returns**db** : `VOSDatabase`

Empty database.

delete_catalog (*name*)

Delete a catalog from database with given name.

Parameters**name** : str

Primary key identifying the catalog.

Raises**MissingCatalog**

If catalog is not found.

delete_catalog_by_url (*url*)Like `delete_catalog()` but using access URL. On multiple matches, all matches are deleted.**classmethod from_json** (*filename*, ***kwargs*)

Create a database of VO services from a JSON file.

Example JSON format for Cone Search:

```
{
  "__version__": 1,
  "catalogs" : {
    "My Cone Search": {
      "capabilityClass": "ConeSearch",
      "title": "My Cone Search",
      "url": "http://foo/cgi-bin/search?CAT=bar&",
      ...
    },
    "Another Cone Search": {
      ...
    }
  }
}
```

Parameters**filename** : str

JSON file.

kwargs : dictKeywords accepted by `get_readable_fileobj()`.**Returns****db** : `VOSDatabase`

Database from given file.

classmethod from_registry (*registry_url*, *timeout=60*, ***kwargs*)

Create a database of VO services from VO registry URL.

This is described in detail in *Building the Database from Registry*, except for the `validate_XXX` keys that are added by the validator itself.

Parameters**registry_url** : str

URL of VO registry that returns a VO Table. For example, see `astropy.vo.validator.validate.CS_MSTR_LIST`. Pedantic is automatically set to `False` for parsing.

timeout : number

Temporarily set `astropy.utils.data.REMOTE_TIMEOUT` to this value to avoid time out error while reading the entire registry.

kwargs : dict

Keywords accepted by `get_readable_fileobj()`.

Returns

db : `VOSDatabase`

Database from given registry.

Raises

VOSError

Invalid VO registry.

get_catalog (*name*)

Get one catalog of given name.

Parameters

name : str

Primary key identifying the catalog.

Returns

obj : `VOSCatalog`

Raises

MissingCatalog

If catalog is not found.

get_catalog_by_url (*url*)

Like `get_catalog()` but using access URL look-up. On multiple matches, only first match is returned.

get_catalogs ()

Iterator to get all catalogs.

get_catalogs_by_url (*url*)

Like `get_catalogs()` but using access URL look-up.

list_catalogs (*pattern=None, sort=True*)

List catalog names.

Parameters

pattern : str or `None`

If given string is anywhere in a catalog name, it is considered a matching catalog. It accepts patterns as in `fnmatch` and is case-insensitive. By default, all catalogs are returned.

sort : bool

Sort output in alphabetical order. If not sorted, the order depends on dictionary hashing. Default is `True`.

Returns

out_arr : list of str

List of catalog names.

list_catalogs_by_url (*pattern=None, sort=True*)

Like `list_catalogs()` but using access URL.

merge (*other, **kwargs*)

Merge two database together.

Parameters

other : `VOSDatabase`

The other database to merge.

kwargs : dict

Keywords accepted by `add_catalog()`.

Returns

db : `VOSDatabase`

Merged database.

Raises

VOSError

Invalid database or incompatible version.

to_json (*filename, clobber=False*)

Write database content to a JSON file.

Parameters

filename : str

JSON file.

clobber : bool

Overwrite existing file?

Raises

OSError

File exists.

astropy.vo.client.conesearch Module

Support VO Simple Cone Search capabilities.

Functions

<code>conesearch(center, radius[, verb])</code>	Perform Cone Search and returns the result of the first successful query.
<code>search_all(*args, **kwargs)</code>	Perform Cone Search and returns the results of all successful queries.
<code>list_catalogs(**kwargs)</code>	Return the available Cone Search catalogs as a list of strings.
<code>predict_search(url, *args, **kwargs)</code>	Predict the run time needed and the number of objects for a Cone Search for the given address.
<code>conesearch_timer(*args, **kwargs)</code>	Time a single Cone Search using <code>astropy.utils.timer.timefunc</code> with a single query.

conesearch

`astropy.vo.client.conesearch.conesearch` (*center, radius, verb=1, **kwargs*)

Perform Cone Search and returns the result of the first successful query.

Parameters**center** : `SkyCoord`, `BaseCoordinateFrame`, or sequence of length 2

Position of the center of the cone to search. It may be specified as an object from the *Astronomical Coordinate Systems* (`astropy.coordinates`) package, or as a length 2 sequence. If a sequence, it is assumed to be (RA, DEC) in the ICRS coordinate frame, given in decimal degrees.

radius : float or `Quantity`

Radius of the cone to search:

- If float is given, it is assumed to be in decimal degrees.
- If astropy quantity is given, it is internally converted to degrees.

verb : {1, 2, 3}

Verbosity indicating how many columns are to be returned in the resulting table. Support for this parameter by a Cone Search service implementation is optional. If the service supports the parameter:

- 1.Return the bare minimum number of columns that the provider considers useful in describing the returned objects.
- 2.Return a medium number of columns between the minimum and maximum (inclusive) that are considered by the provider to most typically useful to the user.
- 3.Return all of the columns that are available for describing the objects.

If not supported, the service should ignore the parameter and always return the same columns for every request.

catalog_db

May be one of the following, in order from easiest to use to most control:

- None**: A database of `astropy.vo.Config.conesearch_dbname` catalogs is downloaded from `astropy.vo.Config.vos_baseurl`. The first catalog in the database to successfully return a result is used.
- catalog name**: A name in the database of `astropy.vo.Config.conesearch_dbname` catalogs at `astropy.vo.Config.vos_baseurl` is used. For a list of acceptable names, use `list_catalogs()`.
- url**: The prefix of a URL to a IVOA Service for `astropy.vo.Config.conesearch_dbname`. Must end in either '?' or '&'.
- VOSCatalog** object: A specific catalog manually downloaded and selected from the database (see *General VO Services Access*).
- Any of the above 3 options combined in a list, in which case they are tried in order.

pedantic : bool or `None`

When `True`, raise an error when the file violates the spec, otherwise issue a warning. Warnings may be controlled using `warnings` module. When not provided, uses the configuration setting `astropy.io.votable.Config.pedantic`, which defaults to `False`.

verbose : bool

Verbose output.

cache : bool

Use caching for VO Service database. Access to actual VO websites referenced by the database still needs internet connection.

Returns

obj : `astropy.io.votable.tree.Table`

First table from first successful VO service request.

Raises

ConeSearchError

When invalid inputs are passed into Cone Search.

VOSError

If VO service request fails.

search_all

`astropy.vo.client.conesearch.search_all(*args, **kwargs)`

Perform Cone Search and returns the results of all successful queries.

Warning: Could potentially take up significant run time and computing resources.

Parameters

args, kwargs :

Arguments and keywords accepted by `conesearch()`.

Returns

all_results : dict of `astropy.io.votable.tree.Table` objects

A dictionary of tables from successful VO service requests, with keys being the access URLs. If none is successful, an empty dictionary is returned.

Raises

ConeSearchError

When invalid inputs are passed into Cone Search.

list_catalogs

`astropy.vo.client.conesearch.list_catalogs(**kwargs)`

Return the available Cone Search catalogs as a list of strings. These can be used for the `catalog_db` argument to `conesearch()`.

Parameters

cache : bool

Use caching for VO Service database. Access to actual VO websites referenced by the database still needs internet connection.

verbose : bool

Show download progress bars.

pattern : str or `None`

If given string is anywhere in a catalog name, it is considered a matching catalog. It accepts patterns as in `fnmatch` and is case-insensitive. By default, all catalogs are returned.

sort : bool

Sort output in alphabetical order. If not sorted, the order depends on dictionary hashing.
Default is `True`.

Returns

arr : list of str

List of catalog names.

predict_search

`astropy.vo.client.conesearch.predict_search(url, *args, **kwargs)`

Predict the run time needed and the number of objects for a Cone Search for the given access URL, position, and radius.

Run time prediction uses `astropy.utils.timer.RunTimePredictor`. Baseline searches are done with starting and ending radii at 0.05 and 0.5 of the given radius, respectively.

Extrapolation on good data uses least-square straight line fitting, assuming linear increase of search time and number of objects with radius, which might not be accurate for some cases. If there are less than 3 data points in the fit, it fails.

Warnings (controlled by `warnings`) are given when:

1. Fitted slope is negative.
2. Any of the estimated results is negative.
3. Estimated run time exceeds `astropy.utils.data.Conf.remote_timeout`.

Note: If `verbose=True`, extra log info will be provided. But unlike `conesearch_timer()`, timer info is suppressed.

If `plot=True`, plot will be displayed. Plotting uses `matplotlib`.

The predicted results are just *rough* estimates.

Prediction is done using `conesearch()`. Prediction for `AsyncConeSearch` is not supported.

Parameters

url : str

Cone Search access URL to use.

args, kwargs : see `conesearch()`

Extra keyword `plot` is allowed and only used by this function and not `conesearch()`.

Returns

t_est : float

Estimated time in seconds needed for the search.

n_est : int

Estimated number of objects the search will yield.

Raises

AssertionError

If prediction fails.

ConeSearchError

If input parameters are invalid.

VOSError

If VO service request fails.

conesearch_timer

`astropy.vo.client.conesearch.conesearch_timer(*args, **kwargs)`

Time a single Cone Search using `astropy.utils.timer.timefunc` with a single try and a verbose timer.

Parameters

`args, kwargs` : see `conesearch()`

Returns

`t` : float

Run time in seconds.

`obj` : `astropy.io.votable.tree.Table`

First table from first successful VO service request.

Classes

<code>AsyncConeSearch(*args, **kwargs)</code>	Perform a Cone Search asynchronously and returns the result of the first successful query.
<code>AsyncSearchAll(*args, **kwargs)</code>	Perform a Cone Search asynchronously, storing all results instead of just the result from first successful query.

AsyncConeSearch

class `astropy.vo.client.conesearch.AsyncConeSearch(*args, **kwargs)`

Bases: `astropy.vo.client.async.AsyncBase`

Perform a Cone Search asynchronously and returns the result of the first successful query.

Note: See `AsyncBase` for more details.

Parameters

`args, kwargs` : see `conesearch()`

Examples

```
>>> from astropy import coordinates as coord
>>> from astropy import units as u
>>> c = coord.ICRS(6.0223 * u.degree, -72.0814 * u.degree)
>>> async_search = conesearch.AsyncConeSearch(
...     c, 0.5 * u.degree,
...     catalog_db='The PMM USNO-A1.0 Catalogue (Monet 1997) 1')
```

Check search status:

```
>>> async_search.running()
True
>>> async_search.done()
False
```

Get search results after a 30-second wait (not to be confused with `astropy.utils.data.Conf.remote_timeout` that governs individual Cone Search queries). If search is still not done after 30 seconds, `TimeoutError` is raised. Otherwise, Cone Search result is returned and can be manipulated as in *Simple Cone Search Examples*. If no `timeout` keyword given, it waits until completion:

```
>>> async_result = async_search.get(timeout=30)
>>> cone_arr = async_result.array.data
>>> cone_arr.size
36184
```

AsyncSearchAll

class `astropy.vo.client.conesearch.AsyncSearchAll(*args, **kwargs)`

Bases: `astropy.vo.client.async.AsyncBase`

Perform a Cone Search asynchronously, storing all results instead of just the result from first successful query.

Note: See `AsyncBase` for more details.

Parameters

`args, kwargs` : see `search_all()`

Examples

```
>>> from astropy import coordinates as coord
>>> from astropy import units as u
>>> c = coord.ICRS(6.0223 * u.degree, -72.0814 * u.degree)
>>> async_searchall = conesearch.AsyncSearchAll(c, 0.5 * u.degree)
```

Check search status:

```
>>> async_search.running()
True
>>> async_search.done()
False
```

Get a dictionary of all search results after a 30-second wait (not to be confused with `astropy.utils.data.Conf.remote_timeout` that governs individual Cone Search queries). If search is still not done after 30 seconds, `TimeoutError` is raised. Otherwise, a dictionary is returned and can be manipulated as in *Simple Cone Search Examples*. If no `timeout` keyword given, it waits until completion:

```
>>> async_allresults = async_search.get(timeout=30)
>>> all_catalogs = list(async_allresults)
>>> first_cone_arr = async_allresults[all_catalogs[0]].array.data
>>> first_cone_arr.size
36184
```

astropy.vo.client.async Module

Asynchronous VO service requests.

Classes

`AsyncBase(func, *args, **kwargs)` Base class for asynchronous VO service requests using `concurrent.futures.ThreadPoolExecutor`.

AsyncBase

class `astropy.vo.client.async.AsyncBase` (*func*, *args, **kwargs)

Bases: `object`

Base class for asynchronous VO service requests using `concurrent.futures.ThreadPoolExecutor`.

Service request will be forced to run in silent mode by setting `verbose=False`. Warnings are controlled by `warnings` module.

Note: Methods of the attributes can be accessed directly, with priority given to `executor`.

Parameters

func : function

The function to run.

args, kwargs

Arguments and keywords accepted by the service request function to be called asynchronously.

Attributes

<code>executor</code>	(<code>concurrent.futures.ThreadPoolExecutor</code>) Executor running the function on single thread.
<code>future</code>	(<code>concurrent.futures.Future</code>) Asynchronous execution created by <code>executor</code> .

Methods Summary

`get([timeout])` Get result, if available, then shut down thread.

Methods Documentation

get (*timeout=None*)

Get result, if available, then shut down thread.

Parameters

timeout : int or float

Wait the given amount of time in seconds before obtaining result. If not given, wait indefinitely until function is done.

Returns

result

Result returned by the function.

Raises

Exception

Errors raised by `concurrent.futures.Future`.

astropy.vo.client.exceptions Module

Exceptions related to Virtual Observatory (VO).

Classes

<code>BaseVOError</code>	Base class for VO exceptions.
<code>VOSError</code>	General VO service exception.
<code>MissingCatalog</code>	VO catalog is missing.
<code>DuplicateCatalogName</code>	VO catalog of the same title already exists.
<code>DuplicateCatalogURL</code>	VO catalog of the same access URL already exists.
<code>InvalidAccessURL</code>	Invalid access URL.
<code>ConeSearchError</code>	General Cone Search exception.

BaseVOError

exception `astropy.vo.client.exceptions.BaseVOError`
Base class for VO exceptions.

VOSError

exception `astropy.vo.client.exceptions.VOSError`
General VO service exception.

MissingCatalog

exception `astropy.vo.client.exceptions.MissingCatalog`
VO catalog is missing.

DuplicateCatalogName

exception `astropy.vo.client.exceptions.DuplicateCatalogName`
VO catalog of the same title already exists.

DuplicateCatalogURL

exception `astropy.vo.client.exceptions.DuplicateCatalogURL`
VO catalog of the same access URL already exists.

InvalidAccessURL

exception `astropy.vo.client.exceptions.InvalidAccessURL`
Invalid access URL.

ConeSearchError

exception `astropy.vo.client.exceptions.ConeSearchError`
General Cone Search exception.

Class Inheritance Diagram

astropy.vo.validator Module

Classes

`Conf` Configuration parameters for `astropy.vo.validator`.

Conf

class `astropy.vo.validator.Conf`

Bases: `astropy.config.ConfigNamespace`

Configuration parameters for `astropy.vo.validator`.

Attributes Summary

<code>conesearch_master_list</code>	URL to the cone search services master list for validation.
<code>conesearch_urls</code>	A list of conesearch URLs to validate.
<code>noncritical_warnings</code>	A list of <code>astropy.io.votable</code> warning codes that are considered non-critical.

Attributes Documentation

`conesearch_master_list`

URL to the cone search services master list for validation.

`conesearch_urls`

A list of conesearch URLs to validate.

`noncritical_warnings`

A list of `astropy.io.votable` warning codes that are considered non-critical.

Class Inheritance Diagram

astropy.vo.validator.validate Module

Validate VO Services.

Functions

`check_conesearch_sites(*args, **kwargs)` Validate Cone Search Services.

`check_conesearch_sites`

`astropy.vo.validator.validate.check_conesearch_sites(*args, **kwargs)`

Validate Cone Search Services.

Note: URLs are unescaped prior to validation.

Only check queries with `<testQuery>` parameters. Does not perform meta-data and erroneous queries.

Parameters

destdir : str, optional

Directory to store output files. Will be created if does not exist. Existing files with these names will be deleted or replaced:

- conesearch_good.json
- conesearch_warn.json
- conesearch_exception.json
- conesearch_error.json

verbose : bool, optional

Print extra info to log.

parallel : bool, optional

Enable multiprocessing.

url_list : list of string, optional

Only check these access URLs against `astropy.vo.validator.Conf.conesearch_master_list` and ignore the others, which will not appear in output files. By default, check those in `astropy.vo.validator.Conf.conesearch_urls`. If `None`, check everything.

Raises

IOError

Invalid destination directory.

timeout

URL request timed out.

ValidationMultiprocessingError

Multiprocessing failed.

astropy.vo.validator.inspect Module

Inspect results from `astropy.vo.validator.validate`.

Classes

`ConeSearchResults([cache, verbose])` A class to store Cone Search validation results.

ConeSearchResults

class `astropy.vo.validator.inspect.ConeSearchResults` (*cache=False, verbose=True*)

Bases: `object`

A class to store Cone Search validation results.

Parameters

cache : bool

Read from cache, if available. Default is `False` to ensure the latest data are read.

verbose : bool

Show download progress bars.

Attributes

<code>dbtypes</code>	(list) Cone Search database identifiers.
<code>dfs</code>	(dict) Stores <code>VOSDatabase</code> for each <code>dbtypes</code> .
<code>catkeys</code>	(dict) Stores sorted catalog keys for each <code>dbtypes</code> .

Methods Summary

<code>list_cats(typ[, fout, ignore_noncrit])</code>	List catalogs in given database.
<code>print_cat(key[, fout])</code>	Display a single catalog of given key.
<code>tally([fout])</code>	Tally databases.

Methods Documentation

list_cats (*typ, fout=None, ignore_noncrit=False*)

List catalogs in given database.

Listing contains:

- 1.Catalog key
- 2.Cone search access URL
- 3.Warning codes
- 4.Warning descriptions

Parameters

typ : str

Any value in `self.dtypes`.

fout : output stream

Default is screen output.

ignore_noncrit : bool

Exclude warnings in `astropy.vo.validator.Conf.noncritical_warnings`.
This is useful to see why a catalog failed validation.

print_cat (*key, fout=None*)

Display a single catalog of given key.

If not found, nothing is written out.

Parameters

key : str

Catalog key.

fout : output stream

Default is screen output.

tally (*fout=None*)

Tally databases.

Parameters

fout : output stream

Default is screen output.

astropy.vo.validator.exceptions Module

Exceptions related to Virtual Observatory (VO) validation.

Classes

<code>BaseVOValidationError</code>	Base class for VO validation exceptions.
<code>ValidationMultiprocessingError</code>	Validation using multiprocessing failed.

BaseVOValidationError

exception `astropy.vo.validator.exceptions.BaseVOValidationError`

Base class for VO validation exceptions.

ValidationMultiprocessingError

exception `astropy.vo.validator.exceptions.ValidationMultiprocessingError`

Validation using multiprocessing failed.

Class Inheritance Diagram

21.1.2 SAMP (Simple Application Messaging Protocol (`astropy.vo.samp`))

Introduction

`astropy.vo.samp` is an IVOA SAMP (Simple Application Messaging Protocol) messaging system implementation in Python. It provides classes to easily:

1. instantiate one or multiple Hubs;
2. interface an application or script to a running Hub;
3. create and manage a SAMP client.

`astropy.vo.samp` provides also a stand-alone program `samp_hub` capable to instantiate a persistent hub.

SAMP is a protocol that is used by a number of other tools such as [TOPCAT](#), [SAO Ds9](#), and [Aladin](#), which means that it is possible to send and receive data to and from these tools. The `astropy.vo.samp` package also supports the ‘web profile’ for SAMP, which means that it can be used to communicate with web SAMP clients. See the `sampjs` library examples for more details.

The following classes are available in `astropy.vo.samp`:

- `SAMPHubServer`, which is used to instantiate a hub server that clients can then connect to.
- `SAMPHubProxy`, which is used to connect to an existing hub (including hubs started from other applications such as [TOPCAT](#)).
- `SAMPClient`, which is used to create a SAMP client

- `SAMPIntegratedClient`, which is the same as `SAMPClient` except that it has a self-contained `SAMPHubProxy` to provide a simpler user interface.

Using `astropy.vo.samp`

Starting and stopping a SAMP hub server

There are several ways you can start up a SAMP hub:

Using an existing hub You can start up another application that includes a hub, such as [TOPCAT](#), [SAO Ds9](#), or [Aladin Desktop](#).

Using the command-line hub utility You can make use of the `samp_hub` command-line utility, which is included in Astropy:

```
$ samp_hub
```

To get more help on available options for `samp_hub`:

```
$ samp_hub -h
```

To stop the server, you can simply press control-C.

Starting a hub programmatically (advanced) You can start up a hub by creating a `SAMPHubServer` instance and starting it, either from the interactive Python prompt, or from a Python script:

```
>>> from astropy.vo.samp import SAMPHubServer
>>> hub = SAMPHubServer()
>>> hub.start()
```

You can then stop the hub by calling:

```
>>> hub.stop()
```

However, this method is generally not recommended for average users because it does not work correctly when web SAMP clients try and connect. Instead, this should be reserved for developers who want to embed a SAMP hub in a GUI for example. For more information, see [Embedding a SAMP hub in a GUI](#).

Sending/receiving tables and images over SAMP

In the following examples, we make use of:

- [TOPCAT](#), which is a tool to explore tabular data.
- [SAO Ds9](#), which is an image visualization tool, which can also overplot catalogs.
- [Aladin Desktop](#), which is another tool that can visualize images and catalogs.

[TOPCAT](#) and [Aladin](#) will run a SAMP Hub if none is found, so for the following examples you can either start up one of these applications first, or you can start up the `astropy.vo.samp` hub. You can start this using the following command:

```
$ samp_hub
```

Sending a table to TOPCAT and Ds9 The easiest way to send a VO table to TOPCAT is to make use of the `SAMPIntegratedClient` class. Once TOPCAT is open, then first instantiate a `SAMPIntegratedClient` instance and connect to the hub:

```
>>> from astropy.vo.samp import SAMPIntegratedClient
>>> client = SAMPIntegratedClient()
>>> client.connect()
```

Next, we have to set up a dictionary that contains details about the table to send. This should include `url`, which is the URL to the file, and `name`, which is a human-readable name for the table. The URL can be a local URL (starting with `file:///`):

```
>>> params = {}
>>> params["url"] = 'file:///Users/tom/Desktop/aj285677t3_votable.xml'
>>> params["name"] = "Robitaille et al. (2008), Table 3"
```

Note: To construct a local URL, you can also make use of `urlparse` as follows:

```
>>> import urlparse
>>> params["url"] = urlparse.urljoin('file:', os.path.abspath("aj285677t3_votable.xml"))
```

Now we can set up the message itself. This includes the type of message (here we use `table.load.votable` which indicates that a VO table should be loaded, and the details of the table that we set above:

```
>>> message = {}
>>> message["samp.mtype"] = "table.load.votable"
>>> message["samp.params"] = params
```

Finally, we can broadcast this to all clients that are listening for `table.load.votable` messages using `notify_all()`:

```
>>> client.notify_all(message)
```

The above message will actually be broadcast to all applications connected via SAMP. For example, if we open *SAO Ds9* in addition to TOPCAT, and we run the above command, both applications will load the table. We can use the `get_registered_clients()` method to find all the clients connected to the hub:

```
>>> client.get_registered_clients()
['hub', 'c1', 'c2']
```

These IDs don't mean much, but we can find out more using:

```
>>> client.get_metadata('c1')
{'author.affiliation': 'Astrophysics Group, Bristol University',
 'author.email': 'm.b.taylor@bristol.ac.uk',
 'author.name': 'Mark Taylor',
 'home.page': 'http://www.starlink.ac.uk/topcat/',
 'samp.description.text': 'Tool for OPERations on Catalogues And Tables',
 'samp.documentation.url': 'http://127.0.0.1:2525/doc/sun253/index.html',
 'samp.icon.url': 'http://127.0.0.1:2525/doc/images/tc_sok.gif',
 'samp.name': 'topcat',
 'topcat.version': '4.0-1'}
```

We can see that `c1` is the TOPCAT client. We can now re-send the data, but this time only to TOPCAT, using the `notify()` method:

```
>>> client.notify('c1', message)
```

Once finished, we should make sure we disconnect from the hub:

```
>>> client.disconnect()
```

Receiving a table from TOPCAT To receive a table from TOPCAT, we have to set up a client that listens for messages from the hub. As before, we instantiate a `SAMPIntegratedClient` instance and connect to the hub:

```
>>> from astropy.vo.samp import SAMPIntegratedClient
>>> client = SAMPIntegratedClient()
>>> client.connect()
```

We now set up a receiver class which will handle any received message. We need to take care to write handlers for both notifications and calls (the difference between the two being that calls expect a reply):

```
>>> class Receiver(object):
...     def __init__(self, client):
...         self.client = client
...         self.received = False
...     def receive_call(self, private_key, sender_id, msg_id, mtype, params, extra):
...         self.params = params
...         self.received = True
...         self.client.reply(msg_id, {"samp.status": "samp.ok", "samp.result": {}})
...     def receive_notification(self, private_key, sender_id, mtype, params, extra):
...         self.params = params
...         self.received = True
```

and we instantiate it:

```
>>> r = Receiver(client)
```

We can now use the `bind_receive_call()` and `bind_receive_notification()` methods to tell our receiver to listen to all `table.load.votable` messages:

```
>>> client.bind_receive_call("table.load.votable", r.receive_call)
>>> client.bind_receive_notification("table.load.votable", r.receive_notification)
```

We can now check that the message has not been received yet:

```
>>> r.received
False
```

Let's now broadcast the table from TOPCAT. After a few seconds, we can try and check again if the message has been received:

```
>>> r.received
True
```

Success! The table URL should now be available in `r.params['url']`, so we can do:

```
>>> from astropy.table import Table
>>> t = Table.read(r.params['url'])
Downloading http://127.0.0.1:2525/dynamic/4/t12.vot [Done]
>>> t
```

	col1	col2	col3	col4	col5	col6	col7	col8	col9	col10
SSTGLMC	G000.0046+01.1431	0.0046	1.1432	265.2992	-28.3321	6.67	5.04	6.89	5.22	N
SSTGLMC	G000.0106-00.7315	0.0106	-0.7314	267.1274	-29.3063	7.18	6.07	nan	5.17	Y
SSTGLMC	G000.0110-01.0237	0.0110	-1.0236	267.4151	-29.4564	8.32	6.30	8.34	6.32	N
...										

As before, we should remember to disconnect from the hub once we are done:

```
>>> client.disconnect()
```

The following is a full example of a script that can be used to receive and read a table. It includes a loop that waits until the message is received, and reads the table once it has:

```
import time

from astropy.vo.samp import SAMPIntegratedClient
from astropy.table import Table

# Instantiate the client and connect to the hub
client=SAMPIntegratedClient()
client.connect()

# Set up a receiver class
class Receiver(object):
    def __init__(self, client):
        self.client = client
        self.received = False
    def receive_call(self, private_key, sender_id, msg_id, mtype, params, extra):
        self.params = params
        self.received = True
        self.client.reply(msg_id, {"samp.status": "samp.ok", "samp.result": {}})
    def receive_notification(self, private_key, sender_id, mtype, params, extra):
        self.params = params
        self.received = True

# Instantiate the receiver
r = Receiver(client)

# Listen for any instructions to load a table
client.bind_receive_call("table.load.votable", r.receive_call)
client.bind_receive_notification("table.load.votable", r.receive_notification)

# We now run the loop to wait for the message in a try/finally block so that if
# the program is interrupted e.g. by control-C, the client terminates
# gracefully.

try:

    # We test every 0.1s to see if the hub has sent a message
    while True:
        time.sleep(0.1)
        if r.received:
            t = Table.read(r.params['url'])
            break

finally:

    client.disconnect()

# Print out table
print t
```

Sending an image to Ds9 and Aladin As for tables, the easiest way to send a FITS image over SAMP is to make use of the `SAMPIntegratedClient` class. Once Aladin or Ds9 are open, then first instantiate a `SAMPIntegratedClient` instance and connect to the hub as before:

```
>>> from astropy.vo.samp import SAMPIntegratedClient
>>> client = SAMPIntegratedClient()
>>> client.connect()
```

Next, we have to set up a dictionary that contains details about the image to send. This should include `url`, which is the URL to the file, and `name`, which is a human-readable name for the table. The URL can be a local URL (starting with `file:///`):

```
>>> params = {}
>>> params["url"] = 'file:///Users/tom/Desktop/MSX_E.fits'
>>> params["name"] = "MSX Band E Image of the Galactic Center"
```

See [Sending a table to TOPCAT and Ds9](#) for an example of how to construct local URLs more easily. Now we can set up the message itself. This includes the type of message (here we use `image.load.fits` which indicates that a FITS image should be loaded, and the details of the table that we set above:

```
>>> message = {}
>>> message["samp.mtype"] = "image.load.fits"
>>> message["samp.params"] = params
```

Finally, we can broadcast this to all clients that are listening for `table.load.votable` messages:

```
>>> client.notify_all(message)
```

As for [Sending a table to TOPCAT and Ds9](#), the `notify_all()` method will broadcast the image to all listening clients, and as for tables it is possible to instead use the `notify()` method to send it to a specific client.

Once finished, we should make sure we disconnect from the hub:

```
>>> client.disconnect()
```

Receiving a table from Ds9 or Aladin Receiving images over SAMP is identical to [Receiving a table from TOPCAT](#), with the exception that the message type should be `image.load.fits` instead of `table.load.votable`. Once the URL has been received, the FITS image can be opened with:

```
>>> from astropy.io import fits
>>> fits.open(r.params['url'])
```

Communication between integrated clients objects

As shown in [Sending/receiving tables and images over SAMP](#), the `SAMPIntegratedClient` class can be used to communicate with other SAMP-enabled tools such as [TOPCAT](#), [SAO Ds9](#), or [Aladin Desktop](#).

In this section, we look at how we can set up two `SAMPIntegratedClient` instances and communicate between them.

First, start up a SAMP hub as described in [Starting and stopping a SAMP hub server](#).

Next, we create two clients and connect them to the hub:

```
>>> client1 = samp.SAMPIntegratedClient(name="Client 1", description="Test Client 1",
...                                     metadata = {"client1.version":"0.01"})
>>> client2 = samp.SAMPIntegratedClient(name="Client 2", description="Test Client 2",
...                                     metadata = {"client2.version":"0.25"})
>>> client1.connect()
>>> client2.connect()
```

We now define functions to call when receiving a notification, call or response:

```
>>> def test_receive_notification(private_key, sender_id, mtype, params, extra):
...     print("Notification:", private_key, sender_id, mtype, params, extra)

>>> def test_receive_call(private_key, sender_id, msg_id, mtype, params, extra):
...     print("Call:", private_key, sender_id, msg_id, mtype, params, extra)
...     client1.ereply(msg_id, SAMP_STATUS_OK, result = {"txt": "printed"})

>>> def test_receive_response(private_key, sender_id, msg_id, response):
...     print("Response:", private_key, sender_id, msg_id, response)
```

We subscribe client 1 to "samp.*" and "samp.app.*" and bind them to the related functions:

```
>>> client1.bind_receive_notification("samp.app.*", test_receive_notification)
>>> client1.bind_receive_call("samp.app.*", test_receive_call)
```

We now bind message tags received by client 2 to suitable functions:

```
>>> client2.bind_receive_response("my-dummy-print", test_receive_response)
>>> client2.bind_receive_response("my-dummy-print-specific", test_receive_response)
```

We are now ready to test out the clients and callback functions. Client 2 notifies all clients using the "samp.app.echo" message type via the hub:

```
>>> client2.enotify_all("samp.app.echo", txt="Hello world!")
['cli#2']
Notification: 0d7f4500225981c104a197c7666a8e4e cli#2 samp.app.echo {'txt':
'Hello world!'} {'host': 'antigone.lambrate.inaf.it', 'user': 'unknown'}
```

We can also find a dictionary giving the clients that would currently receive samp.app.echo messages:

```
>>> print(client2.getSubscribedClients("samp.app.echo"))
{'cli#2': {}}
```

Client 2 calls all clients with the "samp.app.echo" message type using "my-dummy-print" as a message-tag:

```
>>> print(client2.call_all("my-dummy-print",
...                       {"samp.mtype": "samp.app.echo",
...                       "samp.params": {"txt": "Hello world!"}}))
{'cli#1': 'msg#1;;cli#hub;;cli#2;;my-dummy-print'}
Call: 8c8eb53178cb95e168ab17ec4eac2353 cli#2
msg#1;;cli#hub;;cli#2;;my-dummy-print samp.app.echo {'txt': 'Hello world!'}
{'host': 'antigone.lambrate.inaf.it', 'user': 'unknown'}
Response: d0a28636321948ccff45edaf40888c54 cli#1 my-dummy-print
{'samp.status': 'samp.ok', 'samp.result': {'txt': 'printed'}}
```

Client 2 then calls client 1 using the "samp.app.echo" message type, tagging the message as "my-dummy-print-specific":

```
>>> try:
...     print(client2.call(client1.getPublicId(),
...                       "my-dummy-print-specific",
...                       {"samp.mtype": "samp.app.echo",
...                       "samp.params": {"txt": "Hello client 1!"}}))
... except SAMPProxyError as e:
...     print("Error ({0}): {1}".format(e.faultCode, e.faultString))
msg#2;;cli#hub;;cli#2;;my-dummy-print-specific
Call: 8c8eb53178cb95e168ab17ec4eac2353 cli#2
msg#2;;cli#hub;;cli#2;;my-dummy-print-specific samp.app.echo {'txt': 'Hello
Cli 1!'} {'host': 'antigone.lambrate.inaf.it', 'user': 'unknown'}
```

```
Response: d0a28636321948ccff45edaf40888c54 cli#1 my-dummy-print-specific
{'samp.status': 'samp.ok', 'samp.result': {'txt': 'printed'}}
```

We can now define a function called to test synchronous calls:

```
>>> def test_receive_sync_call(private_key, sender_id, msg_id, mtype, params, extra):
...     import time
...     print("SYNC Call:", sender_id, msg_id, mtype, params, extra)
...     time.sleep(2)
...     client1.reply(msg_id, {"samp.status": SAMP_STATUS_OK,
...                            "samp.result": {"txt": "printed sync"}})
... 
```

We now bind the `samp.test` message type to `test_receive_sync_call`:

```
>>> client1.bind_receive_call("samp.test", test_receive_sync_call)
>>> try:
...     # Sync call
...     print(client2.call_and_wait(client1.getPublicId(),
...                                 {"samp.mtype": "samp.test",
...                                  "samp.params": {"txt": "Hello SYNCRO client 1!"}},
...                                 "10"))
... except SAMPProxyError as e:
...     # If timeout expires than a SAMPProxyError is returned
...     print("Error ({0}): {1}".format(e.faultCode, e.faultString))
SYNC Call: cli#2 msg#3;;cli#hub;;cli#2;;sampy::sync::call samp.test {'txt':
'Hello SYNCRO Cli 1!'} {'host': 'antigone.lambrate.inaf.it', 'user':
'unknown'}
{'samp.status': 'samp.ok', 'samp.result': {'txt': 'printed sync'}}
```

Finally, we disconnect the clients from the hub at the end:

```
>>> client1.disconnect()
>>> client2.disconnect()
```

Embedding a SAMP hub in a GUI

Overview If you wish to embed a SAMP hub in your Python GUI tool, you will need to start the hub programmatically using:

```
from astropy.vo.samp import SAMPHubServer
hub = SAMPHubServer()
hub.start()
```

This launches the hub in a thread and is non-blocking. If you are not interested in connections from web SAMP clients, then you can simply use:

```
from astropy.vo.samp import SAMPHubServer
hub = SAMPHubServer(web_profile=False)
hub.start()
```

and this should be all you need to do. However, if you want to keep the web profile active, there is an additional consideration, which is that when a web SAMP client connects, you will need to ask the user whether they accept the connection (for security reasons). By default, the confirmation message is a text-based message in the terminal, but if you have a GUI tool, you will instead likely want to open a GUI dialog.

To do this, you will need to define a class that handles the dialog, and you should then pass an **instance** of the class to `SAMPHubServer` (not the class itself). This class should inherit from `astropy.vo.samp.WebProfileDialog` and add the following:

1. It should have a GUI timer callback that periodically calls `WebProfileDialog.handle_queue` (available as `self.handle_queue`).
2. Implement a `show_dialog` method to display a consent dialog. It should take the following arguments:
 - `samp_name`: The name of the application making the request.
 - `details`: A dictionary of details about the client making the request. The only key in this dictionary required by the SAMP standard is `samp.name` which gives the name of the client making the request.
 - `client`: A hostname, port pair containing the client address.
 - `origin`: A string containing the origin of the request.
3. Based on the user response, the `show_dialog` should call `WebProfileDialog.consent` or `WebProfileDialog.reject`. This may, in some cases, be the result of another GUI callback.

Example of embedding a SAMP hub in a Tk application The following code is a full example of a simple Tk application that watches for web SAMP connections and opens the appropriate dialog:

```
import Tkinter as tk
import tkMessageBox

from astropy.vo.samp import SAMPHubServer
from astropy.vo.samp.hub import WebProfileDialog

MESSAGE = """
A Web application which declares to be

Name: {name}
Origin: {origin}

is requesting to be registered with the SAMP Hub. Pay attention
that if you permit its registration, such application will acquire
all current user privileges, like file read/write.

Do you give your consent?
"""

class TkWebProfileDialog(WebProfileDialog):
    def __init__(self, root):
        self.root = root
        self.wait_for_dialog()

    def wait_for_dialog(self):
        self.handle_queue()
        self.root.after(100, self.wait_for_dialog)

    def show_dialog(self, samp_name, details, client, origin):
        text = MESSAGE.format(name=samp_name, origin=origin)

        response = tkMessageBox.askyesno(
            'SAMP Hub', text,
            default=tkMessageBox.NO)

        if response:
            self.consent()
        else:
            self.reject()
```

```

# Start up Tk application
root = tk.Tk()
tk.Label(root, text="Example SAMP Tk application",
         font=("Helvetica", 36), justify=tk.CENTER).pack(pady=200)
root.geometry("500x500")
root.update()

# Start up SAMP hub
h = SAMPHubServer(web_profile_dialog=TkWebProfileDialog(root))
h.start()

try:
    # Main GUI loop
    root.mainloop()
except KeyboardInterrupt:
    pass

h.stop()

```

If you run the above script, a window will open saying “Example SAMP Tk application”. If you then go to the following page for example:

<http://astrojrs.github.io/sampjs/examples/pinger.html>

and click on the Ping button, you will see the dialog open in the Tk application. Once you click on ‘CONFIRM’, future ‘Ping’ calls will no longer bring up the dialog.

Reference/API

astropy.vo.samp Module

This subpackage provides classes to communicate with other applications via the [Simple Application Messaging Protocol \(SAMP\)](#).

Before integration into Astropy it was known as [SAMPy](#), and was developed by Luigi Paioro (INAF - Istituto Nazionale di Astrofisica).

<code>Conf</code>	Configuration parameters for <code>astropy.vo.samp</code> .
<code>SAMPClient(hub[, name, description, ...])</code>	Utility class which provides facilities to create and manage a SAMP compliant client.
<code>SAMPClientError</code>	SAMP Client exceptions.
<code>SAMPHubError</code>	SAMP Hub exception.
<code>SAMPHubProxy()</code>	Proxy class to simplify the client interaction with a SAMP hub (via the standard SAMP Hub Server).
<code>SAMPHubServer([secret, addr, port, ...])</code>	SAMP Hub Server.
<code>SAMPIntegratedClient([name, description, ...])</code>	A Simple SAMP client.
<code>SAMPMsgReplierWrapper(cli)</code>	Function decorator that allows to automatically grab errors and returned map.
<code>SAMPProxyError(faultCode, faultString, **extra)</code>	SAMP Proxy Hub exception
<code>SAMPWarning</code>	SAMP-specific Astropy warning class
<code>WebProfileDialog</code>	A base class to make writing Web Profile GUI consent dialogs easier.

Classes

Conf

class `astropy.vo.samp.Conf`
Bases: `astropy.config.ConfigNamespace`
Configuration parameters for `astropy.vo.samp`.

Attributes Summary

<code>use_internet</code>	Whether to allow <code>astropy.vo.samp</code> to use the internet, if available.
---------------------------	--

Attributes Documentation

use_internet
Whether to allow `astropy.vo.samp` to use the internet, if available.

SAMPClient

class `astropy.vo.samp.SAMPClient` (*hub*, *name=None*, *description=None*, *metadata=None*,
addr=None, *port=0*, *https=False*, *key_file=None*,
cert_file=None, *cert_reqs=0*, *ca_certs=None*,
ssl_version=None, *callable=True*)

Bases: `object`

Utility class which provides facilities to create and manage a SAMP compliant XML-RPC server that acts as SAMP callable client application.

Parameters

hub : `SAMPHubProxy`

An instance of `SAMPHubProxy` to be used for messaging with the SAMP Hub.

name : str, optional

Client name (corresponding to `samp.name` metadata keyword).

description : str, optional

Client description (corresponding to `samp.description.text` metadata keyword).

metadata : dict, optional

Client application metadata in the standard SAMP format.

addr : str, optional

Listening address (or IP). This defaults to 127.0.0.1 if the internet is not reachable, otherwise it defaults to the host name.

port : int, optional

Listening XML-RPC server socket port. If left set to 0 (the default), the operating system will select a free port.

https : bool, optional

If `True`, set the callable client running on a Secure Sockets Layer (SSL) connection (HTTPS). By default SSL is disabled.

key_file : str, optional

The path to a file containing the private key for SSL connections. If the certificate file (`cert_file`) contains the private key, then `key_file` can be omitted.

cert_file : str, optional

The path to a file which contains a certificate to be used to identify the local side of the secure connection.

cert_reqs : int, optional

Whether a certificate is required from the server side of the connection, and whether it will be validated if provided. It must be one of the three values `ssl.CERT_NONE` (certificates ignored), `ssl.CERT_OPTIONAL` (not required, but validated if provided), or `ssl.CERT_REQUIRED` (required and validated). If the value of this parameter is not `ssl.CERT_NONE`, then the `ca_certs` parameter must point to a file of CA certificates.

ca_certs : str, optional

The path to a file containing a set of concatenated “Certification Authority” certificates, which are used to validate the certificate passed from the Hub end of the connection.

ssl_version : int, optional

Which version of the SSL protocol to use. Typically, the server chooses a particular protocol version, and the client must adapt to the server’s choice. Most of the versions are not interoperable with the other versions. If not specified the default SSL version is `ssl.PROTOCOL_SSLv23`. This version provides the most compatibility with other versions Hub side. Other SSL protocol versions are: `ssl.PROTOCOL_SSLv2`, `ssl.PROTOCOL_SSLv3` and `ssl.PROTOCOL_TLSv1`.

callable : bool, optional

Whether the client can receive calls and notifications. If set to `False`, then the client can send notifications and calls, but can not receive any.

Attributes Summary

<code>is_registered</code>	Whether the client is currently registered.
<code>is_running</code>	Whether the client is currently running.

Methods Summary

<code>bind_receive_call(mtype, function[, ...])</code>	Bind a specific MType call to a function or class method.
<code>bind_receive_message(mtype, function[, ...])</code>	Bind a specific MType to a function or class method, being intended for
<code>bind_receive_notification(mtype, function[, ...])</code>	Bind a specific MType notification to a function or class method.
<code>bind_receive_response(msg_tag, function)</code>	Bind a specific msg-tag response to a function or class method.
<code>declare_metadata([metadata])</code>	Declare the client application metadata supported.
<code>declare_subscriptions([subscriptions])</code>	Declares the MTypes the client wishes to subscribe to, implicitly define
<code>get_private_key()</code>	Return the client private key used for the Standard Profile communicati
<code>get_public_id()</code>	Return public client ID obtained at registration time (<code>samp.self-id</code>)
<code>receive_call(private_key, sender_id, msg_id, ...)</code>	Standard callable client <code>receive_call</code> method.
<code>receive_notification(private_key, sender_id, ...)</code>	Standard callable client <code>receive_notification</code> method.
<code>receive_response(private_key, responder_id, ...)</code>	Standard callable client <code>receive_response</code> method.
<code>register()</code>	Register the client to the SAMP Hub.

<code>start()</code>	Start the client in a separate thread (non-blocking).
<code>stop([timeout])</code>	Stop the client.
<code>unbind_receive_call(mtype[, declare])</code>	Remove from the calls binding table the specified MType and unsubscribe.
<code>unbind_receive_notification(mtype[, declare])</code>	Remove from the notifications binding table the specified MType and unsubscribe.
<code>unbind_receive_response(msg_tag)</code>	Remove from the responses binding table the specified message-tag.
<code>unregister()</code>	Unregister the client from the SAMP Hub.

Attributes Documentation

is_registered

Whether the client is currently registered.

is_running

Whether the client is currently running.

Methods Documentation

bind_receive_call (*mtype, function, declare=True, metadata=None*)

Bind a specific MType call to a function or class method.

The function must be of the form:

```
def my_function_or_method(<self,> private_key, sender_id, msg_id,
                          mtype, params, extra)
```

where `private_key` is the client private-key, `sender_id` is the notification sender ID, `msg_id` is the Hub message-id, `mtype` is the message MType, `params` is the message parameter set (content of "samp.params") and `extra` is a dictionary containing any extra message map entry. The client is automatically declared subscribed to the MType by default.

Parameters

mtype : str

MType to be caught.

function : callable

Application function to be used when `mtype` is received.

declare : bool, optional

Specify whether the client must be automatically declared as subscribed to the MType (see also `declare_subscriptions()`).

metadata : dict, optional

Dictionary containing additional metadata to declare associated with the MType subscribed to (see also `declare_subscriptions()`).

bind_receive_message (*mtype, function, declare=True, metadata=None*)

Bind a specific MType to a function or class method, being intended for a call or a notification.

The function must be of the form:

```
def my_function_or_method(<self,> private_key, sender_id, msg_id,
                          mtype, params, extra)
```

where `private_key` is the client private-key, `sender_id` is the notification sender ID, `msg_id` is the Hub message-id (calls only, otherwise is `None`), `mtype` is the message MType, `params` is the message parameter set (content of `"samp.params"`) and `extra` is a dictionary containing any extra message map entry. The client is automatically declared subscribed to the MType by default.

Parameters

mtype : str

MType to be caught.

function : callable

Application function to be used when `mtype` is received.

declare : bool, optional

Specify whether the client must be automatically declared as subscribed to the MType (see also `declare_subscriptions()`).

metadata : dict, optional

Dictionary containing additional metadata to declare associated with the MType subscribed to (see also `declare_subscriptions()`).

bind_receive_notification (*mtype, function, declare=True, metadata=None*)

Bind a specific MType notification to a function or class method.

The function must be of the form:

```
def my_function_or_method(<self,> private_key, sender_id, mtype,
                          params, extra)
```

where `private_key` is the client private-key, `sender_id` is the notification sender ID, `mtype` is the message MType, `params` is the notified message parameter set (content of `"samp.params"`) and `extra` is a dictionary containing any extra message map entry. The client is automatically declared subscribed to the MType by default.

Parameters

mtype : str

MType to be caught.

function : callable

Application function to be used when `mtype` is received.

declare : bool, optional

Specify whether the client must be automatically declared as subscribed to the MType (see also `declare_subscriptions()`).

metadata : dict, optional

Dictionary containing additional metadata to declare associated with the MType subscribed to (see also `declare_subscriptions()`).

bind_receive_response (*msg_tag, function*)

Bind a specific msg-tag response to a function or class method.

The function must be of the form:

```
def my_function_or_method(<self,> private_key, responder_id,
                          msg_tag, response)
```

where `private_key` is the client private-key, `responder_id` is the message responder ID, `msg_tag` is the message-tag provided at call time and `response` is the response received.

Parameters

msg_tag : str

Message-tag to be caught.

function : callable

Application function to be used when `msg_tag` is received.

declare_metadata (*metadata=None*)

Declare the client application metadata supported.

Parameters

metadata : dict, optional

Dictionary containing the client application metadata as defined in the SAMP definition document. If omitted, then no metadata are declared.

declare_subscriptions (*subscriptions=None*)

Declares the MTypes the client wishes to subscribe to, implicitly defined with the MType binding methods `bind_receive_notification()` and `bind_receive_call()`.

An optional `subscriptions` map can be added to the final map passed to the `declare_subscriptions()` method.

Parameters

subscriptions : dict, optional

Dictionary containing the list of MTypes to subscribe to, with the same format of the `subscriptions` map passed to the `declare_subscriptions()` method.

get_private_key ()

Return the client private key used for the Standard Profile communications obtained at registration time (`samp.private-key`).

Returns

key : str

Client private key.

get_public_id ()

Return public client ID obtained at registration time (`samp.self-id`).

Returns

id : str

Client public ID.

receive_call (*private_key, sender_id, msg_id, message*)

Standard callable client `receive_call` method.

This method is automatically handled when the `bind_receive_call()` method is used to bind distinct operations to MTypes. In case of a customized callable client implementation that inherits from the `SAMPClient` class this method should be overwritten.

Note: When overwritten, this method must always return a string result (even empty).

Parameters

private_key : str

Client private key.

sender_id : str

Sender public ID.

msg_id : str

Message ID received.

message : dict

Received message.

Returns

confirmation : str

Any confirmation string.

receive_notification (*private_key, sender_id, message*)

Standard callable client `receive_notification` method.

This method is automatically handled when the `bind_receive_notification()` method is used to bind distinct operations to MTypes. In case of a customized callable client implementation that inherits from the `SAMPClient` class this method should be overwritten.

Note: When overwritten, this method must always return a string result (even empty).

Parameters

private_key : str

Client private key.

sender_id : str

Sender public ID.

message : dict

Received message.

Returns

confirmation : str

Any confirmation string.

receive_response (*private_key, responder_id, msg_tag, response*)

Standard callable client `receive_response` method.

This method is automatically handled when the `bind_receive_response()` method is used to bind distinct operations to MTypes. In case of a customized callable client implementation that inherits from the `SAMPClient` class this method should be overwritten.

Note: When overwritten, this method must always return a string result (even empty).

Parameters

private_key : str

Client private key.

responder_id : str

Responder public ID.

msg_tag : str
Response message tag.

response : dict
Received response.

Returns

confirmation : str
Any confirmation string.

register ()
Register the client to the SAMP Hub.

start ()
Start the client in a separate thread (non-blocking).
This only has an effect if `callable` was set to `True` when initializing the client.

stop (*timeout=10.0*)
Stop the client.

Parameters

timeout : float
Timeout after which to give up if the client cannot be cleanly shut down.

unbind_receive_call (*mtype, declare=True*)
Remove from the calls binding table the specified MType and unsubscribe the client from it (if required).

Parameters

mtype : str
MType to be removed.

declare : bool
Specify whether the client must be automatically declared as unsubscribed from the MType (see also `declare_subscriptions()`).

unbind_receive_notification (*mtype, declare=True*)
Remove from the notifications binding table the specified MType and unsubscribe the client from it (if required).

Parameters

mtype : str
MType to be removed.

declare : bool
Specify whether the client must be automatically declared as unsubscribed from the MType (see also `declare_subscriptions()`).

unbind_receive_response (*msg_tag*)
Remove from the responses binding table the specified message-tag.

Parameters

msg_tag : str
Message-tag to be removed.

unregister ()
Unregister the client from the SAMP Hub.

SAMPClientError

exception `astropy.vo.samp.SAMPClientError`
SAMP Client exceptions.

SAMPHubError

exception `astropy.vo.samp.SAMPHubError`
SAMP Hub exception.

SAMPHubProxy

class `astropy.vo.samp.SAMPHubProxy`
Bases: `object`

Proxy class to simplify the client interaction with a SAMP hub (via the standard profile).

Attributes Summary

<code>is_connected</code>	Whether the hub proxy is currently connected to a hub.
---------------------------	--

Methods Summary

<code>call(private_key, recipient_id, msg_tag, message)</code>	Proxy to <code>call</code> SAMP Hub method.
<code>call_all(private_key, msg_tag, message)</code>	Proxy to <code>callAll</code> SAMP Hub method.
<code>call_and_wait(private_key, recipient_id, ...)</code>	Proxy to <code>callAndWait</code> SAMP Hub method.
<code>connect([hub, hub_params, key_file, ...])</code>	Connect to the current SAMP Hub.
<code>declare_metadata(private_key, metadata)</code>	Proxy to <code>declareMetadata</code> SAMP Hub method.
<code>declare_subscriptions(private_key, subscriptions)</code>	Proxy to <code>declareSubscriptions</code> SAMP Hub method.
<code>disconnect()</code>	Disconnect from the current SAMP Hub.
<code>get_metadata(private_key, client_id)</code>	Proxy to <code>getMetadata</code> SAMP Hub method.
<code>get_registered_clients(private_key)</code>	Proxy to <code>getRegisteredClients</code> SAMP Hub method.
<code>get_subscribed_clients(private_key, mtype)</code>	Proxy to <code>getSubscribedClients</code> SAMP Hub method.
<code>get_subscriptions(private_key, client_id)</code>	Proxy to <code>getSubscriptions</code> SAMP Hub method.
<code>notify(private_key, recipient_id, message)</code>	Proxy to <code>notify</code> SAMP Hub method.
<code>notify_all(private_key, message)</code>	Proxy to <code>notifyAll</code> SAMP Hub method.
<code>ping()</code>	Proxy to <code>ping</code> SAMP Hub method (Standard Profile only).
<code>register(secret)</code>	Proxy to <code>register</code> SAMP Hub method.
<code>reply(private_key, msg_id, response)</code>	Proxy to <code>reply</code> SAMP Hub method.
<code>set_xmlrpc_callback(private_key, xmlrpc_addr)</code>	Proxy to <code>setXmlrpcCallback</code> SAMP Hub method (Standard Profile only).
<code>unregister(private_key)</code>	Proxy to <code>unregister</code> SAMP Hub method.

Attributes Documentation**is_connected**

Whether the hub proxy is currently connected to a hub.

Methods Documentation

call (*private_key, recipient_id, msg_tag, message*)
Proxy to `call` SAMP Hub method.

call_all (*private_key, msg_tag, message*)

Proxy to callAll SAMP Hub method.

call_and_wait (*private_key, recipient_id, message, timeout*)

Proxy to callAndWait SAMP Hub method.

connect (*hub=None, hub_params=None, key_file=None, cert_file=None, cert_reqs=0, ca_certs=None, ssl_version=None, pool_size=20*)

Connect to the current SAMP Hub.

Parameters

hub : `SAMPHubServer`, optional

The hub to connect to.

hub_params : dict, optional

Optional dictionary containing the lock-file content of the Hub with which to connect. This dictionary has the form {<token-name>: <token-string>, ...}.

key_file : str, optional

The path to a file containing the private key for SSL connections. If the certificate file (`cert_file`) contains the private key, then `key_file` can be omitted.

cert_file : str, optional

The path to a file which contains a certificate to be used to identify the local side of the secure connection.

cert_reqs : int, optional

Whether a certificate is required from the server side of the connection, and whether it will be validated if provided. It must be one of the three values `ssl.CERT_NONE` (certificates ignored), `ssl.CERT_OPTIONAL` (not required, but validated if provided), or `ssl.CERT_REQUIRED` (required and validated). If the value of this parameter is not `ssl.CERT_NONE`, then the `ca_certs` parameter must point to a file of CA certificates.

ca_certs : str, optional

The path to a file containing a set of concatenated “Certification Authority” certificates, which are used to validate the certificate passed from the Hub end of the connection.

ssl_version : int, optional

Which version of the SSL protocol to use. Typically, the server chooses a particular protocol version, and the client must adapt to the server’s choice. Most of the versions are not interoperable with the other versions. If not specified the default SSL version is `ssl.PROTOCOL_SSLv3`. This version provides the most compatibility with other versions server side. Other SSL protocol versions are: `ssl.PROTOCOL_SSLv2`, `ssl.PROTOCOL_SSLv3` and `ssl.PROTOCOL_TLSv1`.

pool_size : int, optional

The number of socket connections opened to communicate with the Hub.

declare_metadata (*private_key, metadata*)

Proxy to declareMetadata SAMP Hub method.

declare_subscriptions (*private_key, subscriptions*)

Proxy to declareSubscriptions SAMP Hub method.

disconnect ()

Disconnect from the current SAMP Hub.

get_metadata (*private_key, client_id*)
Proxy to getMetadata SAMP Hub method.

get_registered_clients (*private_key*)
Proxy to getRegisteredClients SAMP Hub method.

get_subscribed_clients (*private_key, mtype*)
Proxy to getSubscribedClients SAMP Hub method.

get_subscriptions (*private_key, client_id*)
Proxy to getSubscriptions SAMP Hub method.

notify (*private_key, recipient_id, message*)
Proxy to notify SAMP Hub method.

notify_all (*private_key, message*)
Proxy to notifyAll SAMP Hub method.

ping ()
Proxy to ping SAMP Hub method (Standard Profile only).

register (*secret*)
Proxy to register SAMP Hub method.

reply (*private_key, msg_id, response*)
Proxy to reply SAMP Hub method.

set_xmlrpc_callback (*private_key, xmlrpc_addr*)
Proxy to setXmlrpcCallback SAMP Hub method (Standard Profile only).

unregister (*private_key*)
Proxy to unregister SAMP Hub method.

SAMPHubServer

class astropy.vo.samp.**SAMPHubServer** (*secret=None, addr=None, port=0, lockfile=None, timeout=0, client_timeout=0, mode=u'single', label=u'', https=False, key_file=None, cert_file=None, cert_reqs=0, ca_certs=None, ssl_version=None, web_profile=True, web_profile_dialog=None, web_port=21012, pool_size=20*)

Bases: `object`

SAMP Hub Server.

The SSL parameters are usable only if Python has been compiled with SSL support and/or `ssl` module is installed (available by default since Python 2.6).

Parameters

secret : str, optional

The secret code to use for the SAMP lockfile. If none is specified, the `uuid.uuid1()` function is used to generate one.

addr : str, optional

Listening address (or IP). This defaults to 127.0.0.1 if the internet is not reachable, otherwise it defaults to the host name.

port : int, optional

Listening XML-RPC server socket port. If left set to 0 (the default), the operating system will select a free port.

lockfile : str, optional

Custom lockfile name.

timeout : int, optional

Hub inactivity timeout. If `timeout > 0` then the Hub automatically stops after an inactivity period longer than `timeout` seconds. By default `timeout` is set to 0 (Hub never expires).

client_timeout : int, optional

Client inactivity timeout. If `client_timeout > 0` then the Hub automatically unregisters the clients which result inactive for a period longer than `client_timeout` seconds. By default `client_timeout` is set to 0 (clients never expire).

mode : str, optional

Defines the Hub running mode. If `mode` is `'single'` then the Hub runs using the standard `.samp` lock-file, having a single instance for user desktop session. Otherwise, if `mode` is `'multiple'`, then the Hub runs using a non-standard lock-file, placed in `.samp-1` directory, of the form `samp-hub-<UUID>`, where `<UUID>` is a unique UUID assigned to the hub.

label : str, optional

A string used to label the Hub with a human readable name. This string is written in the lock-file assigned to the `hub.label` token.

https : bool, optional

Set the Hub running on a Secure Sockets Layer connection (HTTPS). By default SSL is disabled.

key_file : str, optional

The path to a file containing the private key for SSL connections. If the certificate file (`cert_file`) contains the private key, then `key_file` can be omitted.

cert_file : str, optional

The path to a file containing a certificate to be used to identify the local side of the secure connection.

cert_reqs : int, optional

The parameter `cert_reqs` specifies whether a certificate is required from the client side of the connection, and whether it will be validated if provided. It must be one of the three values `ssl.CERT_NONE` (certificates ignored), `ssl.CERT_OPTIONAL` (not required, but validated if provided), or `ssl.CERT_REQUIRED` (required and validated). If the value of this parameter is not `ssl.CERT_NONE`, then the `ca_certs` parameter must point to a file of CA certificates.

ca_certs : str, optional

The path to a file containing a set of concatenated “Certification Authority” certificates, which are used to validate the certificate passed from the Hub end of the connection.

ssl_version : int, optional

The `ssl_version` option specifies which version of the SSL protocol to use. Typically, the server chooses a particular protocol version, and the client must adapt to the server’s choice. Most of the versions are not interoperable with the other versions. If not specified the default SSL version is `ssl.PROTOCOL_SSLv23`.

This version provides the most compatibility with other versions client side. Other SSL protocol versions are: `ssl.PROTOCOL_SSLv2`, `ssl.PROTOCOL_SSLv3` and `ssl.PROTOCOL_TLSv1`.

web_profile : bool, optional

Enables or disables the Web Profile support.

web_profile_dialog : class, optional

Allows a class instance to be specified using `web_profile_dialog` to replace the terminal-based message with e.g. a GUI pop-up. Two `queue.Queue` instances will be added to the instance as attributes `queue_request` and `queue_result`. When a request is received via the `queue_request` queue, the pop-up should be displayed, and a value of `True` or `False` should be added to `queue_result` depending on whether the user accepted or refused the connection.

web_port : int, optional

The port to use for web SAMP. This should not be changed except for testing purposes, since web SAMP should always use port 21012.

pool_size : int, optional

The number of socket connections opened to communicate with the clients.

Attributes Summary

<code>id</code>	The unique hub ID.
<code>is_running</code>	Return an information concerning the Hub running status.
<code>params</code>	The hub parameters (which are written to the logfile)

Methods Summary

<code>get_mtype_subtypes(mtype)</code>	Return a list containing all the possible wildcarded subtypes of MType.
<code>start([wait])</code>	Start the current SAMP Hub instance and create the lock file.
<code>stop()</code>	Stop the current SAMP Hub instance and delete the lock file.

Attributes Documentation

id

The unique hub ID.

is_running

Return an information concerning the Hub running status.

Returns

running : bool

Is the hub running?

params

The hub parameters (which are written to the logfile)

Methods Documentation

static `get_mtype_subtypes` (*mtype*)

Return a list containing all the possible wildcarded subtypes of MType.

Parameters

mtype : str

MType to be parsed.

Returns

types : list

List of subtypes

Examples

```
>>> from astropy.vo.samp import SAMPHubServer
>>> SAMPHubServer.get_mtype_subtypes("samp.app.ping")
['samp.app.ping', 'samp.app.*', 'samp.*', '*']
```

start (*wait=False*)

Start the current SAMP Hub instance and create the lock file. Hub start-up can be blocking or non blocking depending on the `wait` parameter.

Parameters

wait : bool

If `True` then the Hub process is joined with the caller, blocking the code flow. Usually `True` option is used to run a stand-alone Hub in an executable script. If `False` (default), then the Hub process runs in a separated thread. `False` is usually used in a Python shell.

stop ()

Stop the current SAMP Hub instance and delete the lock file.

SAMPIntegratedClient

class `astropy.vo.samp.SAMPIntegratedClient` (*name=None, description=None, metadata=None, addr=None, port=0, https=False, key_file=None, cert_file=None, cert_reqs=0, ca_certs=None, ssl_version=None, callable=True*)

Bases: `object`

A Simple SAMP client.

This class is meant to simplify the client usage providing a proxy class that merges the `SAMPClient` and `SAMPHubProxy` functionalities in a simplified API.

Parameters

name : str, optional

Client name (corresponding to `samp.name` metadata keyword).

description : str, optional

Client description (corresponding to `samp.description.text` metadata keyword).

metadata : dict, optional

Client application metadata in the standard SAMP format.

addr : str, optional

Listening address (or IP). This defaults to 127.0.0.1 if the internet is not reachable, otherwise it defaults to the host name.

port : int, optional

Listening XML-RPC server socket port. If left set to 0 (the default), the operating system will select a free port.

https : bool, optional

If `True`, set the callable client running on a Secure Sockets Layer (SSL) connection (HTTPS). By default SSL is disabled.

key_file : str, optional

The path to a file containing the private key for SSL connections. If the certificate file (`cert_file`) contains the private key, then `key_file` can be omitted.

cert_file : str, optional

The path to a file which contains a certificate to be used to identify the local side of the secure connection.

cert_reqs : int, optional

Whether a certificate is required from the server side of the connection, and whether it will be validated if provided. It must be one of the three values `ssl.CERT_NONE` (certificates ignored), `ssl.CERT_OPTIONAL` (not required, but validated if provided), or `ssl.CERT_REQUIRED` (required and validated). If the value of this parameter is not `ssl.CERT_NONE`, then the `ca_certs` parameter must point to a file of CA certificates.

ca_certs : str, optional

The path to a file containing a set of concatenated “Certification Authority” certificates, which are used to validate the certificate passed from the Hub end of the connection.

ssl_version : int, optional

Which version of the SSL protocol to use. Typically, the server chooses a particular protocol version, and the client must adapt to the server’s choice. Most of the versions are not interoperable with the other versions. If not specified the default SSL version is `ssl.PROTOCOL_SSLv23`. This version provides the most compatibility with other versions Hub side. Other SSL protocol versions are: `ssl.PROTOCOL_SSLv2`, `ssl.PROTOCOL_SSLv3` and `ssl.PROTOCOL_TLSv1`.

callable : bool, optional

Whether the client can receive calls and notifications. If set to `False`, then the client can send notifications and calls, but can not receive any.

Attributes Summary

`is_connected` Testing method to verify the client connection with a running Hub.

Methods Summary

<code>bind_receive_call(mtype, function[, ...])</code>	Bind a specific MType call to a function or class method.
<code>bind_receive_message(mtype, function[, ...])</code>	Bind a specific MType to a function or class method, being intended for
<code>bind_receive_notification(mtype, function[, ...])</code>	Bind a specific MType notification to a function or class method.
<code>bind_receive_response(msg_tag, function)</code>	Bind a specific msg-tag response to a function or class method.
<code>call(recipient_id, msg_tag, message)</code>	Proxy to <code>call</code> SAMP Hub method.
<code>call_all(msg_tag, message)</code>	Proxy to <code>callAll</code> SAMP Hub method.
<code>call_and_wait(recipient_id, message, timeout)</code>	Proxy to <code>callAndWait</code> SAMP Hub method.
<code>connect([hub, hub_params, key_file, ...])</code>	Connect with the current or specified SAMP Hub, start and register the
<code>declare_metadata(metadata)</code>	Proxy to <code>declareMetadata</code> SAMP Hub method.
<code>declare_subscriptions([subscriptions])</code>	Declares the MTypes the client wishes to subscribe to, implicitly define
<code>disconnect()</code>	Unregister the client from the current SAMP Hub, stop the client and d
<code>ecall(recipient_id, msg_tag, mtype, **params)</code>	Easy to use version of <code>call()</code> .
<code>ecall_all(msg_tag, mtype, **params)</code>	Easy to use version of <code>call_all()</code> .
<code>ecall_and_wait(recipient_id, mtype, timeout, ...)</code>	Easy to use version of <code>call_and_wait()</code> .
<code>enotify(recipient_id, mtype, **params)</code>	Easy to use version of <code>notify()</code> .
<code>enotify_all(mtype, **params)</code>	Easy to use version of <code>notify_all()</code> .
<code>ereply(msg_id, status[, result, error])</code>	Easy to use version of <code>reply()</code> .
<code>get_metadata(client_id)</code>	Proxy to <code>getMetadata</code> SAMP Hub method.
<code>get_private_key()</code>	Return the client private key used for the Standard Profile communicati
<code>get_public_id()</code>	Return public client ID obtained at registration time (<code>samp.self-id</code>)
<code>get_registered_clients()</code>	Proxy to <code>getRegisteredClients</code> SAMP Hub method.
<code>get_subscribed_clients(mtype)</code>	Proxy to <code>getSubscribedClients</code> SAMP Hub method.
<code>get_subscriptions(client_id)</code>	Proxy to <code>getSubscriptions</code> SAMP Hub method.
<code>notify(recipient_id, message)</code>	Proxy to <code>notify</code> SAMP Hub method.
<code>notify_all(message)</code>	Proxy to <code>notifyAll</code> SAMP Hub method.
<code>ping()</code>	Proxy to <code>ping</code> SAMP Hub method (Standard Profile only).
<code>receive_call(private_key, sender_id, msg_id, ...)</code>	Standard callable client <code>receive_call</code> method.
<code>receive_notification(private_key, sender_id, ...)</code>	Standard callable client <code>receive_notification</code> method.
<code>receive_response(private_key, responder_id, ...)</code>	Standard callable client <code>receive_response</code> method.
<code>reply(msg_id, response)</code>	Proxy to <code>reply</code> SAMP Hub method.
<code>unbind_receive_call(mtype[, declare])</code>	Remove from the calls binding table the specified MType and unsubs
<code>unbind_receive_notification(mtype[, declare])</code>	Remove from the notifications binding table the specified MType and u
<code>unbind_receive_response(msg_tag)</code>	Remove from the responses binding table the specified message-tag.

Attributes Documentation

`is_connected`

Testing method to verify the client connection with a running Hub.

Returns

`is_connected` : bool

True if the client is connected to a Hub, False otherwise.

Methods Documentation

`bind_receive_call` (*mtype, function, declare=True, metadata=None*)

Bind a specific MType call to a function or class method.

The function must be of the form:

```
def my_function_or_method(<self,> private_key, sender_id, msg_id,
                          mtype, params, extra)
```

where `private_key` is the client private-key, `sender_id` is the notification sender ID, `msg_id` is the Hub message-id, `mtype` is the message MType, `params` is the message parameter set (content of `"samp.params"`) and `extra` is a dictionary containing any extra message map entry. The client is automatically declared subscribed to the MType by default.

Parameters

mtype : str

MType to be caught.

function : callable

Application function to be used when `mtype` is received.

declare : bool, optional

Specify whether the client must be automatically declared as subscribed to the MType (see also `declare_subscriptions()`).

metadata : dict, optional

Dictionary containing additional metadata to declare associated with the MType subscribed to (see also `declare_subscriptions()`).

bind_receive_message (*mtype, function, declare=True, metadata=None*)

Bind a specific MType to a function or class method, being intended for a call or a notification.

The function must be of the form:

```
def my_function_or_method(<self,> private_key, sender_id, msg_id,
                        mtype, params, extra)
```

where `private_key` is the client private-key, `sender_id` is the notification sender ID, `msg_id` is the Hub message-id (calls only, otherwise is `None`), `mtype` is the message MType, `params` is the message parameter set (content of `"samp.params"`) and `extra` is a dictionary containing any extra message map entry. The client is automatically declared subscribed to the MType by default.

Parameters

mtype : str

MType to be caught.

function : callable

Application function to be used when `mtype` is received.

declare : bool, optional

Specify whether the client must be automatically declared as subscribed to the MType (see also `declare_subscriptions()`).

metadata : dict, optional

Dictionary containing additional metadata to declare associated with the MType subscribed to (see also `declare_subscriptions()`).

bind_receive_notification (*mtype, function, declare=True, metadata=None*)

Bind a specific MType notification to a function or class method.

The function must be of the form:

```
def my_function_or_method(<self,> private_key, sender_id, mtype,
                        params, extra)
```

where `private_key` is the client private-key, `sender_id` is the notification sender ID, `mtype` is the message MType, `params` is the notified message parameter set (content of "samp.params") and `extra` is a dictionary containing any extra message map entry. The client is automatically declared subscribed to the MType by default.

Parameters**mtype** : str

MType to be caught.

function : callableApplication function to be used when `mtype` is received.**declare** : bool, optionalSpecify whether the client must be automatically declared as subscribed to the MType (see also `declare_subscriptions()`).**metadata** : dict, optionalDictionary containing additional metadata to declare associated with the MType subscribed to (see also `declare_subscriptions()`).**bind_receive_response** (*msg_tag, function*)Bind a specific `msg_tag` response to a function or class method.

The function must be of the form:

```
def my_function_or_method(<self,> private_key, responder_id,
                          msg_tag, response)
```

where `private_key` is the client private-key, `responder_id` is the message responder ID, `msg_tag` is the message-tag provided at call time and `response` is the response received.

Parameters**msg_tag** : str

Message-tag to be caught.

function : callableApplication function to be used when `msg_tag` is received.**call** (*recipient_id, msg_tag, message*)Proxy to `call` SAMP Hub method.**call_all** (*msg_tag, message*)Proxy to `callAll` SAMP Hub method.**call_and_wait** (*recipient_id, message, timeout*)Proxy to `callAndWait` SAMP Hub method.**connect** (*hub=None, hub_params=None, key_file=None, cert_file=None, cert_reqs=0, ca_certs=None, ssl_version=None, pool_size=20*)

Connect with the current or specified SAMP Hub, start and register the client.

Parameters**hub** : `SAMPHubServer`, optional

The hub to connect to.

hub_params : dict, optionalOptional dictionary containing the lock-file content of the Hub with which to connect. This dictionary has the form `{<token-name>: <token-string>, ...}`.

key_file : str, optional

The path to a file containing the private key for SSL connections. If the certificate file (`cert_file`) contains the private key, then `key_file` can be omitted.

cert_file : str, optional

The path to a file which contains a certificate to be used to identify the local side of the secure connection.

cert_reqs : int, optional

Whether a certificate is required from the server side of the connection, and whether it will be validated if provided. It must be one of the three values `ssl.CERT_NONE` (certificates ignored), `ssl.CERT_OPTIONAL` (not required, but validated if provided), or `ssl.CERT_REQUIRED` (required and validated). If the value of this parameter is not `ssl.CERT_NONE`, then the `ca_certs` parameter must point to a file of CA certificates.

ca_certs : str, optional

The path to a file containing a set of concatenated “Certification Authority” certificates, which are used to validate the certificate passed from the Hub end of the connection.

ssl_version : int, optional

Which version of the SSL protocol to use. Typically, the server chooses a particular protocol version, and the client must adapt to the server’s choice. Most of the versions are not interoperable with the other versions. If not specified the default SSL version is `ssl.PROTOCOL_SSLv3`. This version provides the most compatibility with other versions server side. Other SSL protocol versions are: `ssl.PROTOCOL_SSLv2`, `ssl.PROTOCOL_SSLv3` and `ssl.PROTOCOL_TLSv1`.

pool_size : int, optional

The number of socket connections opened to communicate with the Hub.

declare_metadata (*metadata*)

Proxy to `declareMetadata` SAMP Hub method.

declare_subscriptions (*subscriptions=None*)

Declares the MTypes the client wishes to subscribe to, implicitly defined with the MType binding methods `bind_receive_notification()` and `bind_receive_call()`.

An optional `subscriptions` map can be added to the final map passed to the `declare_subscriptions()` method.

Parameters

subscriptions : dict, optional

Dictionary containing the list of MTypes to subscribe to, with the same format of the `subscriptions` map passed to the `declare_subscriptions()` method.

disconnect ()

Unregister the client from the current SAMP Hub, stop the client and disconnect from the Hub.

ecall (*recipient_id, msg_tag, mtype, **params*)

Easy to use version of `call()`.

This is a proxy to `call` method that allows to send a call message in a simplified way.

Note that reserved `extra_kws` keyword is a dictionary with the special meaning of being used to add extra keywords, in addition to the standard `samp.mtype` and `samp.params`, to the message sent.

Parameters**recipient_id** : str

Recipient ID

msg_tag : str

Message tag to use

mtype : str

MType to be sent

params : dict of set of keywords

Variable keyword set which contains the list of parameters for the specified MType.

Examples

```
>>> from astropy.vo.samp import SAMPIntegratedClient
>>> cli = SAMPIntegratedClient()
>>> ...
>>> msgid = cli.ecall("abc", "xyz", "samp.msg.progress",
...                 txt = "initialization", percent = "10",
...                 extra_kws = {"my.extra.info": "just an example"})
```

ecall_all (*msg_tag, mtype, **params*)Easy to use version of `call_all()`.This is a proxy to `callAll` method that allows to send the call message in a simplified way.Note that reserved `extra_kws` keyword is a dictionary with the special meaning of being used to add extra keywords, in addition to the standard `samp.mtype` and `samp.params`, to the message sent.**Parameters****msg_tag** : str

Message tag to use

mtype : str

MType to be sent

params : dict of set of keywords

Variable keyword set which contains the list of parameters for the specified MType.

Examples

```
>>> from astropy.vo.samp import SAMPIntegratedClient
>>> cli = SAMPIntegratedClient()
>>> ...
>>> msgid = cli.ecall_all("xyz", "samp.msg.progress",
...                     txt = "initialization", percent = "10",
...                     extra_kws = {"my.extra.info": "just an example"})
```

ecall_and_wait (*recipient_id, mtype, timeout, **params*)Easy to use version of `call_and_wait()`.This is a proxy to `callAndWait` method that allows to send the call message in a simplified way.

Note that reserved `extra_kws` keyword is a dictionary with the special meaning of being used to add extra keywords, in addition to the standard `samp.mtype` and `samp.params`, to the message sent.

Parameters

recipient_id : str

Recipient ID

mtype : str

MType to be sent

timeout : str

Call timeout in seconds

params : dict of set of keywords

Variable keyword set which contains the list of parameters for the specified MType.

Examples

```
>>> from astropy.vo.samp import SAMPIntegratedClient
>>> cli = SAMPIntegratedClient()
>>> ...
>>> cli.ecall_and_wait("xyz", "samp.msg.progress", "5",
...                   txt = "initialization", percent = "10",
...                   extra_kws = {"my.extra.info": "just an example"})
```

enotify (*recipient_id*, *mtype*, ****params**)

Easy to use version of `notify()`.

This is a proxy to `notify` method that allows to send the notification message in a simplified way.

Note that reserved `extra_kws` keyword is a dictionary with the special meaning of being used to add extra keywords, in addition to the standard `samp.mtype` and `samp.params`, to the message sent.

Parameters

recipient_id : str

Recipient ID

mtype : str

the MType to be notified

params : dict or set of keywords

Variable keyword set which contains the list of parameters for the specified MType.

Examples

```
>>> from astropy.vo.samp import SAMPIntegratedClient
>>> cli = SAMPIntegratedClient()
>>> ...
>>> cli.enotify("samp.msg.progress", msgid = "xyz", txt = "initialization",
...           percent = "10", extra_kws = {"my.extra.info": "just an example"})
```

enotify_all (*mtype*, ****params**)

Easy to use version of `notify_all()`.

This is a proxy to `notifyAll` method that allows to send the notification message in a simplified way.

Note that reserved `extra_kws` keyword is a dictionary with the special meaning of being used to add extra keywords, in addition to the standard `samp.mtype` and `samp.params`, to the message sent.

Parameters

mtype : str

MType to be notified.

params : dict or set of keywords

Variable keyword set which contains the list of parameters for the specified MType.

Examples

```
>>> from astropy.vo.samp import SAMPIntegratedClient
>>> cli = SAMPIntegratedClient()
>>> ...
>>> cli.enotify_all("samp.msg.progress", txt = "initialization",
...               percent = "10",
...               extra_kws = {"my.extra.info": "just an example"})
```

ereply (*msg_id*, *status*, *result=None*, *error=None*)

Easy to use version of `reply()`.

This is a proxy to `reply` method that allows to send a reply message in a simplified way.

Parameters

msg_id : str

Message ID to which reply.

status : str

Content of the `samp.status` response keyword.

result : dict

Content of the `samp.result` response keyword.

error : dict

Content of the `samp.error` response keyword.

Examples

```
>>> from astropy.vo.samp import SAMPIntegratedClient, SAMP_STATUS_ERROR
>>> cli = SAMPIntegratedClient()
>>> ...
>>> cli.ereply("abd", SAMP_STATUS_ERROR, result={},
...           error={"samp.errortxt": "Test error message"})
```

get_metadata (*client_id*)

Proxy to `getMetadata` SAMP Hub method.

get_private_key ()

Return the client private key used for the Standard Profile communications obtained at registration time (`samp.private-key`).

Returns

key : str

Client private key.

get_public_id()

Return public client ID obtained at registration time (`samp.self-id`).

Returns

id : str

Client public ID.

get_registered_clients()

Proxy to `getRegisteredClients` SAMP Hub method.

This returns all the registered clients, excluding the current client.

get_subscribed_clients(*mtype*)

Proxy to `getSubscribedClients` SAMP Hub method.

get_subscriptions(*client_id*)

Proxy to `getSubscriptions` SAMP Hub method.

notify(*recipient_id, message*)

Proxy to `notify` SAMP Hub method.

notify_all(*message*)

Proxy to `notifyAll` SAMP Hub method.

ping()

Proxy to `ping` SAMP Hub method (Standard Profile only).

receive_call(*private_key, sender_id, msg_id, message*)

Standard callable client `receive_call` method.

This method is automatically handled when the `bind_receive_call()` method is used to bind distinct operations to MTypes. In case of a customized callable client implementation that inherits from the `SAMPClient` class this method should be overwritten.

Note: When overwritten, this method must always return a string result (even empty).

Parameters

private_key : str

Client private key.

sender_id : str

Sender public ID.

msg_id : str

Message ID received.

message : dict

Received message.

Returns

confirmation : str

Any confirmation string.

receive_notification(*private_key, sender_id, message*)

Standard callable client `receive_notification` method.

This method is automatically handled when the `bind_receive_notification()` method is used to bind distinct operations to MTypes. In case of a customized callable client implementation that inherits from the `SAMPClient` class this method should be overwritten.

Note: When overwritten, this method must always return a string result (even empty).

Parameters

private_key : str

Client private key.

sender_id : str

Sender public ID.

message : dict

Received message.

Returns

confirmation : str

Any confirmation string.

receive_response (*private_key, responder_id, msg_tag, response*)

Standard callable client `receive_response` method.

This method is automatically handled when the `bind_receive_response()` method is used to bind distinct operations to MTypes. In case of a customized callable client implementation that inherits from the `SAMPClient` class this method should be overwritten.

Note: When overwritten, this method must always return a string result (even empty).

Parameters

private_key : str

Client private key.

responder_id : str

Responder public ID.

msg_tag : str

Response message tag.

response : dict

Received response.

Returns

confirmation : str

Any confirmation string.

reply (*msg_id, response*)

Proxy to `reply` SAMP Hub method.

unbind_receive_call (*mtype, declare=True*)

Remove from the calls binding table the specified MType and unsubscribe the client from it (if required).

Parameters**mtype** : str

MType to be removed.

declare : boolSpecify whether the client must be automatically declared as unsubscribed from the MType (see also `declare_subscriptions()`).**unbind_receive_notification** (*mtype, declare=True*)

Remove from the notifications binding table the specified MType and unsubscribe the client from it (if required).

Parameters**mtype** : str

MType to be removed.

declare : boolSpecify whether the client must be automatically declared as unsubscribed from the MType (see also `declare_subscriptions()`).**unbind_receive_response** (*msg_tag*)

Remove from the responses binding table the specified message-tag.

Parameters**msg_tag** : str

Message-tag to be removed.

SAMPMsgReplierWrapper**class** `astropy.vo.samp.SAMPMsgReplierWrapper` (*cli*)Bases: `object`

Function decorator that allows to automatically grab errors and returned maps (if any) from a function bound to a SAMP call (or notify).

Parameters**cli** : `SAMPIntegratedClient` or `SAMPClient`

SAMP client instance. Decorator initialization, accepting the instance of the client that receives the call or notification.

Methods Summary

`__call__(f)`

Methods Documentation`__call__(f)`**SAMPProxyError****exception** `astropy.vo.samp.SAMPProxyError` (*faultCode, faultString, **extra*)

SAMP Proxy Hub exception

SAMPWarning

exception `astropy.vo.samp.SAMPWarning`
SAMP-specific Astropy warning class

WebProfileDialog

class `astropy.vo.samp.WebProfileDialog`
Bases: `object`

A base class to make writing Web Profile GUI consent dialogs easier.

The concrete class must:

1. Poll `handle_queue` periodically, using the timer services of the GUI's event loop. This function will call `self.show_dialog` when a request requires authorization. `self.show_dialog` will be given the arguments:
 - `samp_name`: The name of the application making the request.
 - `details`: A dictionary of details about the client making the request.
 - `client`: A hostname, port pair containing the client address.
 - `origin`: A string containing the origin of the request.
2. Call `consent` or `reject` based on the user's response to the dialog.

Methods Summary

```
consent()
handle_queue()
reject()
```

Methods Documentation

consent ()

handle_queue ()

reject ()

Class Inheritance Diagram

Acknowledgments

This code is adapted from the [SAMPy](#) package written by Luigi Paioro, who has granted the Astropy project permission to use the code under a BSD license.

Other third-party Python packages and tools related to `astropy.vo`:

- [PyVO](#) provides further functionality to discover and query VO services. Its user guide contains a [good introduction](#) to how the VO works.

- [Astroquery](#) is an Astropy affiliated package that provides simply access to specific astronomical web services, many of which do not support the VO protocol.
- [Simple-Cone-Search-Creator](#) shows how to ingest a catalog into a cone search service and serve it in VO standard format using Python (using CSV files and [healpy](#)).

Nuts and bolts of Astropy

CONFIGURATION SYSTEM (ASTROPY . CONFIG)

22.1 Introduction

The astropy configuration system is designed to give users control of various parameters used in astropy or affiliated packages without delving into the source code to make those changes.

Note: The configuration system got a major overhaul in astropy 0.4 as part of APE3. See *Configuration transition* for information about updating code to use the new API.

22.2 Getting Started

The Astropy configuration options are most easily set by modifying the configuration file. It will be automatically generated with all the default values commented out the first time you import Astropy. You can find the exact location by doing:

```
>>> from astropy.config import get_config_dir
>>> get_config_dir()
```

And you should see the location of your configuration directory. The standard scheme generally puts your configuration directory in `$HOME/.astropy/config`, but if you've set the environment variable `XDG_CONFIG_HOME` and the `$XDG_CONFIG_HOME/astropy` directory exists, it will instead be there.

Note: This is a slight variation from the behavior of most Linux applications with respect to `$XDG_CONFIG_HOME`, where the default, if `$XDG_CONFIG_HOME` is not defined, would be to put configuration in `~/.config/astropy`.

Once you've found the configuration file, open it with your favorite editor. It should have all of the sections you might want, with descriptions and the type of the value that is accepted. Feel free to edit this as you wish, and any of these changes will be reflected when you next start Astropy. Or, if you want to see your changes immediately in your current Astropy session, just do:

```
>>> from astropy.config import reload_config
>>> reload_config()
```

Note: If for whatever reason your `$HOME/.astropy` directory is not accessible (i.e., you have astropy running somehow as root but you are not the root user), the best solution is to set the `XDG_CONFIG_HOME` and `XDG_CACHE_HOME` environment variables pointing to directories, and create an `astropy` directory inside each of those. Both the configuration and data download systems will then use those directories and never try to access the `$HOME/.astropy` directory.

22.3 Using `astropy.config`

22.3.1 Accessing Values

By convention, configuration parameters live inside of objects called `conf` at the root of each subpackage. For example, configuration parameters related to data files live in `astropy.utils.data.conf`. This object has properties for getting and setting individual configuration parameters. For instance to get the default URL for astropy remote data do:

```
>>> from astropy.utils.data import conf
>>> conf.dataurl
'http://data.astropy.org/'
```

22.3.2 Changing Values at Run-time

Changing configuration values persistently is done by editing the configuration file as described above. Values can also, however, be modified in an active python session by setting any of the properties on a `conf` object.

For example, if there is a part of your configuration file that looks like:

```
[utils.data]

# URL for astropy remote data site.
dataurl = http://data.astropy.org/

# Time to wait for remote data query (in seconds).
remote_timeout = 3.0
```

You should be able to modify the values at run-time this way:

```
>>> from astropy.utils.data import conf
>>> conf.dataurl
'http://data.astropy.org/'
>>> conf.dataurl = 'http://astropydata.mywebsite.com'
>>> conf.dataurl
'http://astropydata.mywebsite.com'
>>> conf.remote_timeout
3.0
>>> conf.remote_timeout = 4.5
>>> conf.remote_timeout
4.5
```

22.3.3 Reloading Configuration

Instead of modifying the variables in python, you can also modify the configuration files and then reload them. For example, if you modify the configuration file to say:

```
[utils.data]

# URL for astropy remote data site.
dataurl = http://myotherdata.mywebsite.com/

# Time to wait for remote data query (in seconds).
remote_timeout = 6.3
```

And then run the following commands:

```
>>> conf.reload('dataurl')
>>> conf.reload('remote_timeout')
```

This should update the variables with the values from the configuration file:

```
>>> conf.dataurl
'http://myotherdata.mywebsite.com/'
>>> conf.remote_timeout
6.3
```

You can reload all configuration parameters of a `conf` object at once by calling `reload` with no parameters:

```
>>> conf.reload()
```

Or if you want to reload all astropy configuration at once, use the `reload_config` function:

```
>>> config.reload_config('astropy')
```

You can also reset a configuration parameter back to its default value. Note that this is the default value defined in the Python code, and has nothing to do with the configuration file on disk:

```
>>> conf.reset('dataurl')
>>> conf.dataurl
'http://data.astropy.org/'
```

22.3.4 Upgrading astropy

Each time you upgrade to a new major version of astropy, the configuration parameters may have changed.

If you never edited your configuration file, there is nothing for you to do. It will automatically be replaced with a configuration file template for the newly installed version of astropy.

If you did customize your configuration file, it will not be touched. Instead, a new configuration file template will be installed alongside it with the version number in the filename, for example `astropy.0.4.cfg`. You can compare this file to your `astropy.cfg` file to see what needs to be changed or updated.

22.4 Adding new configuration items

Configuration items should be used wherever an option or setting is needed that is either tied to a system configuration or should persist across sessions of astropy or an affiliated package. Options that may affect the results of science calculations should not be configuration items, but should instead be `astropy.utils.state.ScienceState`, so it's possible to reproduce science results without them being affected by configuration parameters set in a particular environment. Admittedly, this is only a guideline, as the precise cases where a configuration item is preferred over, say, a keyword option for a function is somewhat personal preference. It is the preferred form of persistent configuration, however, and astropy packages must all use it (and it is recommended for affiliated packages).

The reference guide below describes the interface for creating a `conf` object with a number of configuration parameters. They should be defined at the top level, i.e. in the `__init__.py` of each subpackage that has configuration items:

```
""" This is the docstring at the beginning of a module
"""
from astropy import config as _config

class Conf(_config.ConfigNamespace):
```

```
"""
Configuration parameters for my subpackage.
"""
some_setting = _config.ConfigItem(
    1, 'Description of some_setting')
another_setting = _config.ConfigItem(
    'string value', 'Description of another_setting')
# Create an instance for the user
conf = Conf()

... implementation ...
def some_func():
    #to get the value of these options, I might do:
    something = conf.some_setting + 2
    return conf.another_setting + ' Also, I added text.'
```

The configuration items also need to be added to the config file template. For astropy, this file is in `astropy/astropy.cfg`. For an affiliated package called, for example, `packagename`, the file is in `packagename/packagename.cfg`. For the example above, the following content would be added to the config file template:

```
[subpackage]
## Description of some_setting
# some_setting = 1

## Description of another_setting
# another_setting = foo
```

Note that the key/value pairs are commented out. This will allow for changing the default values in a future version of astropy without requiring the user to edit their configuration file to take advantage of the new defaults. By convention, the descriptions of each parameter are in comment lines starting with two hash characters (`##`) to distinguish them from commented out key/value pairs.

22.4.1 Item Types and Validation

If not otherwise specified, a `ConfigItem` gets its type from the type of the `defaultvalue` it is given when it is created. The item can only be set to be an object of this type. Hence:

```
some_setting = ConfigItem(1, 'A description.')
...
conf.some_setting = 1.2
```

will fail, because `1.2` is a float and `1` is an int.

Note that if you want the configuration item to be limited to a particular set of options, you should pass in a list as the `defaultvalue` option. The first entry in the list will be taken as the default, and the list as a whole gives all the valid options. For example:

```
an_option = ConfigItem(
    ['a', 'b', 'c'],
    "This option can be 'a', 'b', or 'c'")
...
conf.an_option = 'b' # succeeds
conf.an_option = 'c' # succeeds
conf.an_option = 'd' # fails!
conf.an_option = 6 # fails!
```

Finally, a `ConfigItem` can be explicitly given a type via the `cfgtype` option:

```

an_int_setting = ConfigItem(
    1, 'A description.', cfgtype='integer')
...
conf.an_int_setting = 3      # works fine
conf.an_int_setting = 4.2   # fails!

```

If the default value's type doesn't match `cfgtype`, the `ConfigItem` cannot be created:

```

an_int_setting = ConfigItem(
    4.2, 'A description.', cfgtype='integer')

```

In summary, the default behavior (of automatically determining `cfgtype`) is usually what you want. The main exception is when you want your configuration item to be a list. The default behavior will treat that as a list of *options* unless you explicitly tell it that the `ConfigItem` itself is supposed to be a list:

```

a_list_setting = ConfigItem([1, 2, 3], 'A description.')
a_list_setting = ConfigItem([1, 2, 3], 'A description.', cfgtype='list')

```

Details of all the valid `cfgtype` items can be found in the [validation section of the configobj manual](#). Below is a list of the valid values here for quick reference:

- 'integer'
- 'float'
- 'boolean'
- 'string'
- 'ip_addr'
- 'list'
- 'tuple'
- 'int_list'
- 'float_list'
- 'bool_list'
- 'string_list'
- 'ip_addr_list'
- 'mixed_list'
- 'option'
- 'pass'

22.4.2 Usage Tips

Keep in mind is that `ConfigItem` objects can be changed at runtime by users. So it is always recommended to read their values immediately before use instead of just storing their initial value to some other variable (or used as a default for a function). For example, the following will work, but is incorrect usage:

```

def some_func(val=conf.some_setting):
    return val + 2

```

This works fine as long as the user doesn't change its value during runtime, but if they do, the function won't know about the change:

```
>>> some_func()
3
>>> conf.some_setting = 3
>>> some_func() # naively should return 5, because 3 + 2 = 5
3
```

There are two ways around this. The typical/intended way is:

```
def some_func():
    """
    The `SOME_SETTING` configuration item influences this output
    """
    return conf.some_setting + 2
```

Or, if the option needs to be available as a function parameter:

```
def some_func(val=None):
    """
    If not specified, `val` is set by the `SOME_SETTING` configuration item.
    """
    return (conf.some_setting if val is None else val) + 2
```

22.5 See Also

22.5.1 Configuration transition

This document describes the changes in the configuration system in astropy 0.4 and how to update code to use it.

For users

The config file

If you never edited the configuration file in `~/.astropy/config/astropy.cfg`, there is nothing for you to do. The first time you import astropy 0.4, it will automatically be replaced with the configuration file template for astropy 0.4.

If you did edit the configuration file, it will be left untouched. However, the template for astropy 0.4 will be installed as `~/.astropy/config/astropy.0.4.cfg`. You can manually compare your changes to this file to determine what customizations should be brought over.

Saving

Saving configuration items from Python has been completely removed. Instead, the configuration file must be edited directly.

Renames

The location of the configuration parameters have been simplified, so they always appear in a high-level subpackage of astropy, rather than low-level file names (which were really an implementation detail that shouldn't have been exposed to the user). On the Python side, configuration items always are referenced through a `conf` object at the root of a subpackage.

Some configuration items that affect the results of science calculations have been removed as configuration parameters altogether and converted to science state objects that must be changed from Python code.

The following table lists all of the moves (in alphabetical order by original configuration file location). The old names will continue to work both from Python and the configuration file for the astropy 0.4 release cycle, and will be removed altogether in astropy 0.5.

Table 2

Old config file location	Old Python location
[] unicode_output	UNICODE_OUTPUT
[coordinates.name_resolve] name_resolve_timeout	coordinates.name_resolve.NAME_RESOLVE_TIMEOUT
[coordinates.name_resolve] sesame_url	coordinates.name_resolve.SESAME_URL
[coordinates.name_resolve] sesame_database	coordinates.name_resolve.SESAME_DATABASE
[cosmology.core] default_cosmology	cosmology.core.DEFAULT_COSMOLOGY
[io.fits] enable_record_valued_keyword_cards	io.fits.ENABLE_RECORD_VALUED_KEYWORD_CARDS
[io.fits] extension_name_case_sensitive	io.fits.EXTENSION_NAME_CASE_SENSITIVE
[io.fits] strip_header_whitespace	io.fits.STRIP_HEADER_WHITESPACE
[io.fits] use_memmap	io.fits.USE_MEMMAP
[io.votable.table] pedantic	io.votable.table.PEDANTIC
[logger] log_exceptions	logger.LOG_EXCEPTIONS
[logger] log_file_format	logger.LOG_FILE_FORMAT
[logger] log_file_level	logger.LOG_FILE_LEVEL
[logger] log_file_path	logger.LOG_FILE_PATH
[logger] log_level	logger.LOG_LEVEL
[logger] log_to_file	logger.LOG_TO_FILE
[logger] log_warnings	logger.LOG_WARNINGS
[logger] use_color	logger.USE_COLOR
[nndata.nndata] warn_unsupported_correlated	nndata.nndata.WARN_UNSUPPORTED_CORRELATED
[table.column] auto_colname	table.column.AUTO_COLNAME
[table.jsviewer] jquery_url	table.jsviewer.JQUERY_URL
[table.jsviewer] datatables_url	table.jsviewer.DATATABLES_URL
[table.pprint] max_lines	table.pprint.MAX_LINES
[table.pprint] max_width	table.pprint.MAX_WIDTH
[utils.console] use_color	utils.console.USE_COLOR
[utils.data] compute_hash_block_size	astropy.utils.data.COMPUTE_HASH_BLOCK_SIZE
[utils.data] dataurl	astropy.utils.data.DATAURL
[utils.data] delete_temporary_downloads_at_exit	astropy.utils.data.DELETE_TEMPORARY_DOWNLOADS_AT_EXIT
[utils.data] download_cache_block_size	astropy.utils.data.DOWNLOAD_CACHE_BLOCK_SIZE
[utils.data] download_cache_lock_attempts	astropy.utils.data.DOWNLOAD_CACHE_LOCK_ATTEMPTS
[utils.data] remote_timeout	astropy.utils.data.REMOTE_TIMEOUT
[vo.client.conesearch] conesearch_dbname	vo.client.conesearch.CONESearch_DBNAME
[vo.client.vos_catalog] vos_baseurl	vo.client.vos_catalog.BASEURL
[vo.samp.utils] use_internet	vo.samp.utils.ALLOW_INTERNET
[vo.validator.validate] cs_mstr_list	vo.validator.validate.CS_MSTR_LIST
[vo.validator.validate] cs_urls	vo.validator.validate.CS_URLS
[vo.validator.validate] noncrit_warnings	vo.validator.validate.NONCRIT_WARNINGS

For affiliated package authors

For an affiliated package to support both astropy 0.3 and 0.4, following the astropy 0.3 config instructions should continue to work. Note that saving of configuration items has been removed entirely from astropy 0.4 without a deprecation cycle, so if saving configuration programmatically is important to your package, you may want to consider another method to save that state.

However, by the release of astropy 0.5, the astropy 0.3 config API will no longer work. The following describes how to transition an affiliated package written for astropy 0.3 to support astropy 0.4 and later. It will not be possible to support astropy 0.3, 0.4 and 0.5 simultaneously. Below `pkgname` is the name of your affiliated package.

The automatic generation of configuration files from the `ConfigurationItem` objects that it finds has been removed. Instead, the project should include a hard-coded “template” configuration file in `pkgname/pkgname.cfg`. By convention, and to ease upgrades for end users, all of the values should be commented out. For example:

```
[nndata]
```

```
## Whether to issue a warning if NDData arithmetic is performed with
## uncertainties and the uncertainties do not support the propagation of
## correlated uncertainties.
# warn_unsupported_correlated = True
```

Affiliated packages should transition to using `astropy.config.ConfigItem` objects as members of `astropy.config.ConfigNamespace` subclasses.

For example, the following is an example of the astropy 0.3 and earlier method to define configuration items:

```
from astropy.config import ConfigurationItem

ENABLE_RECORD_VALUED_KEYWORD_CARDS = ConfigurationItem(
    'enabled_record_valued_keyword_cards', True,
    'If True, enable support for record-valued keywords as described by '
    'FITS WCS Paper IV. Otherwise they are treated as normal keywords.')

EXTENSION_NAME_CASE_SENSITIVE = ConfigurationItem(
    'extension_name_case_sensitive', False,
    'If True, extension names (i.e. the EXTNAME keyword) should be '
    'treated as case-sensitive.')
```

The above, converted to the new method, looks like:

```
from astropy import config as _config

class Conf(_config.ConfigNamespace):
    """
    Configuration parameters for `astropy.io.fits`.
    """

    enable_record_valued_keyword_cards = _config.ConfigItem(
        True,
        'If True, enable support for record-valued keywords as described by '
        'FITS WCS Paper IV. Otherwise they are treated as normal keywords.',
        aliases=['astropy.io.fits.enabled_record_valued_keyword_cards'])

    extension_name_case_sensitive = _config.ConfigItem(
        False,
        'If True, extension names (i.e. the ``EXTNAME`` keyword) should be '
        'treated as case-sensitive.')

conf = Conf()
```

Moving/renaming configuration items in Python

`ConfigAlias` objects can be used when a configuration item has been moved from an astropy 0.3-style `ConfigurationItem` to an astropy 0.4-style `ConfigItem` inside of a `ConfigNamespace`.

In the above example, the following adds backward-compatible hooks so the old Python locations of the configuration items will continue to work from user code:

```
ENABLE_RECORD_VALUED_KEYWORD_CARDS = _config.ConfigAlias(
    '0.4', 'ENABLE_RECORD_VALUED_KEYWORD_CARDS',
    'enable_record_valued_keyword_cards')
```

Moving/rename configuration items in the config file

If a configuration item is moved or renamed within the configuration file, the `aliases` kwarg to `ConfigItem` can be used so that the old location will continue to be used as a fallback. For example, if the old location of an item was:

```
[coordinates.name_resolve]
sesame_url = http://somewhere.com
```

One might want to drop the fact that that is implemented in the module `name_resolve` and just store the configuration in `coordinates`:

```
[coordinates]
sesame_url = http://somewhere.com
```

When defining the `ConfigItem` for this entry, the `aliases` kwarg can list the old location(s) of the configuration item:

```
sesame_url = _config.ConfigItem(
    ["http://somewhere.com"],
    """Docstring""",
    aliases=['astropy.coordinates.name_resolve.sesame_url'])
```

Logging system (overview of `astropy.logger`)

22.6 Reference/API

22.6.1 `astropy.config` Module

This module contains configuration and setup utilities for the Astropy project. This includes all functionality related to the affiliated package index.

Functions

<code>get_cache_dir()</code>	Determines the Astropy cache directory name and creates the directory if it doesn't exist.
<code>get_config([packageormod, reload])</code>	Gets the configuration object or section associated with a particular package or module.
<code>get_config_dir([create])</code>	Determines the Astropy configuration directory name and creates the directory if it doesn't exist.
<code>reload_config([packageormod])</code>	Reloads configuration settings from a configuration file for the root package of the request.
<code>save_config([packageormod, filename])</code>	Removed in astropy 0.4.

`get_cache_dir`

```
astropy.config.get_cache_dir()
```

Determines the Astropy cache directory name and creates the directory if it doesn't exist.

This directory is typically `$HOME/.astropy/cache`, but if the `XDG_CACHE_HOME` environment variable is set, it will use that instead.

able is set and the `$XDG_CACHE_HOME/astropy` directory exists, it will be that directory. If neither exists, the former will be created and symlinked to the latter.

Returns

cachedir : str

The absolute path to the cache directory.

get_config

`astropy.config.get_config(packageormod=None, reload=False)`

Gets the configuration object or section associated with a particular package or module.

Parameters

packageormod : str or None

The package for which to retrieve the configuration object. If a string, it must be a valid package name, or if `None`, the package from which this function is called will be used.

reload : bool, optional

Reload the file, even if we have it cached.

Returns

cfgobj : `configobj.ConfigObj` or `configobj.Section`

If the requested package is a base package, this will be the `configobj.ConfigObj` for that package, or if it is a subpackage or module, it will return the relevant `configobj.Section` object.

Raises

RuntimeError

If `packageormod` is `None`, but the package this item is created from cannot be determined.

get_config_dir

`astropy.config.get_config_dir(create=True)`

Determines the Astropy configuration directory name and creates the directory if it doesn't exist.

This directory is typically `$HOME/.astropy/config`, but if the `XDG_CONFIG_HOME` environment variable is set and the `$XDG_CONFIG_HOME/astropy` directory exists, it will be that directory. If neither exists, the former will be created and symlinked to the latter.

Returns

configdir : str

The absolute path to the configuration directory.

reload_config

`astropy.config.reload_config(packageormod=None)`

Reloads configuration settings from a configuration file for the root package of the requested package/module.

This overwrites any changes that may have been made in `ConfigItem` objects. This applies for any items that are based on this file, which is determined by the `root` package of `packageormod` (e.g. `'astropy.cfg'` for the `'astropy.config.configuration'` module).

Parameters**packageormod** : str or NoneThe package or module name - see `get_config` for details.**save_config**`astropy.config.save_config` (*packageormod=None, filename=None*)

Removed in astropy 0.4.

Classes

<code>ConfigAlias</code>	A class that exists to support backward compatibility only.
<code>ConfigItem</code>	A setting and associated value stored in a configuration file.
<code>ConfigNamespace</code>	A namespace of configuration items.
<code>ConfigurationItem</code>	A backward-compatibility layer to support the old <code>ConfigurationItem</code> API.
<code>ConfigurationMissingWarning</code>	A <code>Warning</code> that is issued when the configuration directory cannot be accessed (usually <code>~/.astropy</code>).
<code>InvalidConfigurationItemWarning</code>	A <code>Warning</code> that is issued when the configuration value specified in the astropy config file is not a valid configuration item.

config.ConfigAlias`config.ConfigAlias`

A class that exists to support backward compatibility only.

This is an alias for a `ConfigItem` that has been moved elsewhere. It inherits from `ConfigItem` only because it implements the same interface, not because any of the methods are reused.**Parameters****since** : str

The version in which the configuration item was moved.

old_name : str

The old name of the configuration item. This should be the name of the variable in Python, not in the configuration file.

new_name : str

The new name of the configuration item. This is both the name of the item in Python and in the configuration file (since as of astropy 0.4, those are always the same thing).

old_module : str, optionalA fully-qualified, dot-separated path to the module in which the configuration item used to be defined. If not provided, it is the name of the module in which `ConfigAlias` is called.**new_module** : str, optionalA fully-qualified, dot-separated path to the module in which the configuration item is now defined. If not provided, it is the name of the module in which `ConfigAlias` is called. This string should not contain the `.conf` object. For example, if the new configuration item is in `astropy.conf.use_unicode`, this value only needs to be `'astropy'`.

config.ConfigItem

config.ConfigItem

A setting and associated value stored in a configuration file.

These objects should be created as members of `ConfigNamespace` subclasses, for example:

```
class _Conf(config.ConfigNamespace):
    unicode_output = config.ConfigItem(
        False,
        'Use Unicode characters when outputting values, and writing widgets '
        'to the console.')
conf = _Conf()
```

Parameters

defaultvalue : object, optional

The default value for this item. If this is a list of strings, this item will be interpreted as an ‘options’ value - this item must be one of those values, and the first in the list will be taken as the default value.

description : str or None, optional

A description of this item (will be shown as a comment in the configuration file)

cfgtype : str or None, optional

A type specifier like those used as the *values* of a particular key in a `configspect` file of `configobj`. If None, the type will be inferred from the default value.

module : str or None, optional

The full module name that this item is associated with. The first element (e.g. ‘astropy’ if this is ‘astropy.config.configuration’) will be used to determine the name of the configuration file, while the remaining items determine the section. If None, the package will be inferred from the package within which this object’s initializer is called.

aliases : str, or list of str, optional

The deprecated location(s) of this configuration item. If the config item is not found at the new location, it will be searched for at all of the old locations.

Raises

RuntimeError

If `module` is `None`, but the module this item is created from cannot be determined.

ConfigNamespace

class `astropy.config.ConfigNamespace`

Bases: `object`

A namespace of configuration items. Each subpackage with configuration items should define a subclass of this class, containing `ConfigItem` instances as members.

For example:

```
class Conf(_config.ConfigNamespace):
    unicode_output = _config.ConfigItem(
        False,
        'Use Unicode characters when outputting values, ...')
```

```

use_color = _config.ConfigItem(
    sys.platform != 'win32',
    'When True, use ANSI color escape sequences when ...',
    aliases=['astropy.utils.console.USE_COLOR'])
conf = Conf()

```

Methods Summary

<code>reload([attr])</code>	Reload a configuration item from the configuration file.
<code>reset([attr])</code>	Reset a configuration item to its default.
<code>set_temp(attr, value)</code>	Temporarily set a configuration value.

Methods Documentation

`reload (attr=None)`

Reload a configuration item from the configuration file.

Parameters

attr : str, optional

The name of the configuration parameter to reload. If not provided, reload all configuration parameters.

`reset (attr=None)`

Reset a configuration item to its default.

Parameters

attr : str, optional

The name of the configuration parameter to reload. If not provided, reset all configuration parameters.

`set_temp (attr, value)`

Temporarily set a configuration value.

Parameters

attr : str

Configuration item name

value : object

The value to set temporarily.

Examples

```

>>> import astropy
>>> with astropy.conf.set_temp('use_color', False):
...     pass
...     # console output will not contain color
>>> # console output contains color again...

```

config.ConfigurationItem

config.ConfigurationItem

A backward-compatibility layer to support the old `ConfigurationItem` API. The only difference between this and `ConfigItem` is that this requires an explicit name to be set as the first argument.

ConfigurationMissingWarning

exception astropy.config.ConfigurationMissingWarning

A Warning that is issued when the configuration directory cannot be accessed (usually due to a permissions problem). If this warning appears, configuration items will be set to their defaults rather than read from the configuration file, and no configuration will persist across sessions.

InvalidConfigurationItemWarning

exception astropy.config.InvalidConfigurationItemWarning

A Warning that is issued when the configuration value specified in the astropy configuration file does not match the type expected for that configuration value.

Class Inheritance Diagram

I/O REGISTRY (ASTROPY.IO.REGISTRY)

Note: The I/O registry is only meant to be used directly by users who want to define their own custom readers/writers. Users who want to find out more about what built-in formats are supported by `Table` by default should see *Unified file read/write interface*. No built-in formats are currently defined for `NDData`, but this will be added in future).

23.1 Introduction

The I/O registry is a sub-module used to define the readers/writers available for the `Table` and `NDData` classes.

23.2 Using `astropy.io.registry`

The following example demonstrates how to create a reader for the `Table` class. First, we can create a highly simplistic FITS reader which just reads the data as a structured array:

```
from astropy.table import Table

def fits_table_reader(filename, hdu=1):
    from astropy.io import fits
    data = fits.open(filename)[hdu].data
    return Table(data)
```

and then register it:

```
from astropy.io import registry
registry.register_reader('fits', Table, fits_table_reader)
```

Reader functions can take any arguments except `format` (since this is reserved for `read()`) and should return an instance of the class specified as the second argument of `register_reader` (`Table` in the above case.)

We can then read in a FITS table with:

```
t = Table.read('catalog.fits', format='fits')
```

In practice, it would be nice to have the `read` method automatically identify that this file was a FITS file, so we can construct a function that can recognize FITS files, which we refer to here as an *identifier* function. An identifier function should take a first argument that should be a string which indicates whether the identifier is being called from `read` or `write`, and should then accept arbitrary number of positional and keyword arguments via `*args` and `**kwargs`, which are the arguments passed to `Table.read`. We can write a simplistic function that only looks at filenames (but in practice, this function could even look at the first few bytes of the file for example). The only requirement is that it return a boolean indicating whether the input matches that expected for the format:

```
def fits_identify(origin, *args, **kwargs):
    return isinstance(args[0], basestring) and \
        args[0].lower().split('.')[-1] in ['fits', 'fit']
```

Note: Identifier functions should be prepared for arbitrary input - in particular, the first argument may not be a filename or file object, so it should not assume that this is the case.

We then register this identifier function:

```
registry.register_identifier('fits', Table, fits_identify)
```

And we can then do:

```
t = Table.read('catalog.fits')
```

If multiple formats match the current input, then an exception is raised, and similarly if no format matches the current input. In that case, the format should be explicitly given with the `format=` keyword argument.

Similarly, it is possible to create custom writers. To go with our simplistic FITS reader above, we can write a simplistic FITS writer:

```
def fits_table_writer(table, filename, clobber=False):
    import numpy as np
    from astropy.io import fits
    fits.writeto(filename, np.array(table), clobber=clobber)
```

We then register the writer:

```
io_registry.register_writer('fits', Table, fits_table_writer)
```

And we can then write the file out to a FITS file:

```
t.write('catalog_new.fits', format='fits')
```

If we have registered the identifier as above, we can simply do:

```
t.write('catalog_new.fits')
```

23.3 Reference/API

23.3.1 astropy.io.registry Module

Functions

<code>register_reader(data_format, data_class, ...)</code>	Register a reader function.
<code>register_writer(data_format, data_class, ...)</code>	Register a table writer function.
<code>register_identifier(data_format, data_class, ...)</code>	Associate an identifier function with a specific data type.
<code>identify_format(origin, data_class_required, ...)</code>	
<code>get_reader(data_format, data_class)</code>	
<code>get_writer(data_format, data_class)</code>	
<code>read(cls, *args, **kwargs)</code>	Read in data
<code>write(data, *args, **kwargs)</code>	Write out data
<code>get_formats([data_class])</code>	Get the list of registered I/O formats as a Table.

register_reader

`astropy.io.registry.register_reader` (*data_format*, *data_class*, *function*, *force=False*)

Register a reader function.

Parameters

data_format : str

The data type identifier. This is the string that will be used to specify the data type when reading.

data_class : classobj

The class of the object that the reader produces

function : function

The function to read in a data object.

force : bool

Whether to override any existing function if already present.

register_writer

`astropy.io.registry.register_writer` (*data_format*, *data_class*, *function*, *force=False*)

Register a table writer function.

Parameters

data_format : str

The data type identifier. This is the string that will be used to specify the data type when writing.

data_class : classobj

The class of the object that can be written

function : function

The function to write out a data object.

force : bool

Whether to override any existing function if already present.

register_identifier

`astropy.io.registry.register_identifier` (*data_format*, *data_class*, *identifier*, *force=False*)

Associate an identifier function with a specific data type.

Parameters

data_format : str

The data type identifier. This is the string that is used to specify the data type when reading/writing.

data_class : classobj

The class of the object that can be written

identifier : function

A function that checks the argument specified to `read` or `write` to determine whether the input can be interpreted as a table of type `data_format`. This function should take the following arguments:

- `origin`: A string `read` or `write` identifying whether the file is to be opened for reading or writing.
- `path`: The path to the file.
- `fileobj`: An open file object to read the file's contents, or `None` if the file could not be opened.
- `*args`: A list of positional arguments to the `read` or `write` function.
- `**kwargs`: A list of keyword arguments to the `read` or `write` function.

One or both of `path` or `fileobj` may be `None`. If they are both `None`, the identifier will need to work from `args[0]`.

The function should return `True` if the input can be identified as being of format `data_format`, and `False` otherwise.

force : bool

Whether to override any existing function if already present.

Examples

To set the identifier based on extensions, for formats that take a filename as a first argument, you can do for example:

```
>>> def my_identifier(*args, **kwargs):
...     return (isinstance(args[0], basestring) and
...             args[0].endswith('.tbl'))
>>> register_identifier('ipac', Table, my_identifier)
```

identify_format

`astropy.io.registry.identify_format` (*origin, data_class_required, path, fileobj, args, kwargs*)

get_reader

`astropy.io.registry.get_reader` (*data_format, data_class*)

get_writer

`astropy.io.registry.get_writer` (*data_format, data_class*)

read

```
astropy.io.registry.read(cls, *args, **kwargs)
```

Read in data

The arguments passed to this method depend on the format

write

```
astropy.io.registry.write(data, *args, **kwargs)
```

Write out data

The arguments passed to this method depend on the format

get_formats

```
astropy.io.registry.get_formats(data_class=None)
```

Get the list of registered I/O formats as a Table.

Parameters

data_class : classobj

Filter readers/writer to match data class (default = all classes)

Returns

format_table: Table

Table of available I/O formats

LOGGING SYSTEM

24.1 Overview

The Astropy logging system is designed to give users flexibility in deciding which log messages to show, to capture them, and to send them to a file.

All messages printed by Astropy routines should use the built-in logging facility (normal `print()` calls should only be done by routines that are explicitly requested to print output). Messages can have one of several levels:

- **DEBUG**: Detailed information, typically of interest only when diagnosing problems.
- **INFO**: An message conveying information about the current task, and confirming that things are working as expected
- **WARNING**: An indication that something unexpected happened, and that user action may be required.
- **ERROR**: indicates a more serious issue, including exceptions

By default, only **WARNING** and **ERROR** messages are displayed, and are sent to a log file located at `~/astropy/astropy.log` (if the file is writeable).

24.2 Configuring the logging system

First, import the logger:

```
from astropy import log
```

The threshold level (defined above) for messages can be set with e.g.:

```
log.setLevel('INFO')
```

Color (enabled by default) can be disabled with:

```
log.setColor(False)
```

Warnings from `warnings.warn` can be logged with:

```
log.enable_warnings_logging()
```

which can be disabled with:

```
log.disable_warnings_logging()
```

and exceptions can be included in the log with:

```
log.set_exception_logging()
```

which can be disabled with:

```
log.disable_exception_logging()
```

It is also possible to set these settings from the Astropy configuration file, which also allows an overall log file to be specified. See [Using the configuration file](#) for more information.

24.3 Context managers

In some cases, you may want to capture the log messages, for example to check whether a specific message was output, or to log the messages from a specific section of code to a file. Both of these are possible using context managers.

To add the log messages to a list, first import the logger if you have not already done so:

```
from astropy import log
```

then enclose the code in which you want to log the messages to a list in a `with` statement:

```
with log.log_to_list() as log_list:
    # your code here
```

In the above example, once the block of code has executed, `log_list` will be a Python list containing all the Astropy logging messages that were raised. Note that messages continue to be output as normal.

Similarly, you can output the log messages of a specific section of code to a file using:

```
with log.log_to_file('myfile.log'):
    # your code here
```

which will add all the messages to `myfile.log` (this is in addition to the overall log file mentioned in [Using the configuration file](#)).

While these context managers will include all the messages emitted by the logger (using the global level set by `log.setLevel`), it is possible to filter a subset of these using `filter_level=`, and specifying one of `'DEBUG'`, `'INFO'`, `'WARN'`, `'ERROR'`. Note that if `filter_level` is a lower level than that set via `setLevel`, only messages with the level set by `setLevel` or higher will be included (i.e. `filter_level` is only filtering a subset of the messages normally emitted by the logger).

Similarly, it is possible to filter a subset of the messages by origin by specifying `filter_origin=` followed by a string. If the origin of a message starts with that string, the message will be included in the context manager. For example, `filter_origin='astropy.wcs'` will include only messages emitted in the `astropy.wcs` sub-package.

24.4 Using the configuration file

Options for the logger can be set in the `[config.logging_helper]` section of the Astropy configuration file:

```
[config.logging_helper]
# Threshold for the logging messages. Logging messages that are less severe
# than this level will be ignored. The levels are 'DEBUG', 'INFO', 'WARNING',
# 'ERROR'
log_level = 'INFO'
```

```

# Whether to use color for the level names
use_color = True

# Whether to log warnings.warn calls
log_warnings = False

# Whether to log exceptions before raising them
log_exceptions = False

# Whether to always log messages to a log file
log_to_file = True

# The file to log messages to
log_file_path = '~/.astropy/astropy.log'

# Threshold for logging messages to log_file_path
log_file_level = 'INFO'

# Format for log file entries
log_file_format = '%(asctime)s, %(origin)s, %(levelname)s, %(message)s'

```

24.5 Reference/API

24.5.1 astropy.logger Module

This module defines a logging class based on the built-in logging module

Classes

<code>Conf</code>	Configuration parameters for <code>astropy.logger</code> .
<code>AstropyLogger(name[, level])</code>	This class is used to set up the Astropy logging.
<code>LoggingError</code>	This exception is for various errors that occur in the astropy logger, typically when activating or

Conf

class `astropy.logger.Conf`

Bases: `astropy.config.ConfigNamespace`

Configuration parameters for `astropy.logger`.

Attributes Summary

<code>log_exceptions</code>	Whether to log exceptions before raising them.
<code>log_file_format</code>	Format for log file entries.
<code>log_file_level</code>	Threshold for logging messages to <code>log_file_path</code> .
<code>log_file_path</code>	The file to log messages to.
<code>log_level</code>	Threshold for the logging messages.
<code>log_to_file</code>	Whether to always log messages to a log file.
<code>log_warnings</code>	Whether to log <code>warnings.warn</code> calls.

Attributes Documentation

log_exceptions

Whether to log exceptions before raising them.

log_file_format

Format for log file entries.

log_file_level

Threshold for logging messages to `log_file_path`.

log_file_path

The file to log messages to. When `'`, it defaults to a file `'astropy.log'` in the astropy config directory.

log_level

Threshold for the logging messages. Logging messages that are less severe than this level will be ignored. The levels are `'DEBUG'`, `'INFO'`, `'WARNING'`, `'ERROR'`.

log_to_file

Whether to always log messages to a log file.

log_warnings

Whether to log `warnings.warn` calls.

AstropyLogger

class `astropy.logger.AstropyLogger` (*name*, *level=0*)

Bases: `logging.Logger`

This class is used to set up the Astropy logging.

The main functionality added by this class over the built-in `logging.Logger` class is the ability to keep track of the origin of the messages, the ability to enable logging of `warnings.warn` calls and exceptions, and the addition of colored output and context managers to easily capture messages to a file or list.

Initialize the logger with a name and an optional level.

Methods Summary

<code>disable_color()</code>	Disable colored output
<code>disable_exception_logging()</code>	Disable logging of exceptions Once called, any uncaught exceptions will no longer be logged
<code>disable_warnings_logging()</code>	Disable logging of <code>warnings.warn()</code> calls Once called, any subsequent calls to <code>warnings.warn()</code> will not be logged
<code>enable_color()</code>	Enable colored output
<code>enable_exception_logging()</code>	Enable logging of exceptions Once called, any uncaught exceptions will be emitted
<code>enable_warnings_logging()</code>	Enable logging of <code>warnings.warn()</code> calls Once called, any subsequent calls to <code>warnings.warn()</code> will be logged
<code>exception_logging_enabled()</code>	Determine if the exception-logging mechanism is enabled.
<code>log_to_file(*args, **kwds)</code>	Context manager to temporarily log messages to a file.
<code>log_to_list(*args, **kwds)</code>	Context manager to temporarily log messages to a list.
<code>makeRecord(name, level, pathname, lineno, ...)</code>	
<code>setLevel(level)</code>	Set the logging level of this logger.
<code>warnings_logging_enabled()</code>	

Methods Documentation

`disable_color()`

Disable colorized output

`disable_exception_logging()`

Disable logging of exceptions

Once called, any uncaught exceptions will no longer be emitted by this logger.

This can be re-enabled with `enable_exception_logging`.

`disable_warnings_logging()`

Disable logging of `warnings.warn()` calls

Once called, any subsequent calls to `warnings.warn()` are no longer redirected to this logger.

This can be re-enabled with `enable_warnings_logging`.

`enable_color()`

Enable colorized output

`enable_exception_logging()`

Enable logging of exceptions

Once called, any uncaught exceptions will be emitted with level `ERROR` by this logger, before being raised.

This can be disabled with `disable_exception_logging`.

`enable_warnings_logging()`

Enable logging of `warnings.warn()` calls

Once called, any subsequent calls to `warnings.warn()` are redirected to this logger and emitted with level `WARN`. Note that this replaces the output from `warnings.warn`.

This can be disabled with `disable_warnings_logging`.

`exception_logging_enabled()`

Determine if the exception-logging mechanism is enabled.

Returns

exclog : bool

True if exception logging is on, False if not.

`log_to_file(*args, **kws)`

Context manager to temporarily log messages to a file.

Parameters

filename : str

The file to log messages to.

filter_level : str

If set, any log messages less important than `filter_level` will not be output to the file. Note that this is in addition to the top-level filtering for the logger, so if the logger has level `'INFO'`, then setting `filter_level` to `INFO` or `DEBUG` will have no effect, since these messages are already filtered out.

filter_origin : str

If set, only log messages with an origin starting with `filter_origin` will be output to the file.

Notes

By default, the logger already outputs log messages to a file set in the Astropy configuration file. Using this context manager does not stop log messages from being output to that file, nor does it stop log messages from being printed to standard output.

Examples

The context manager is used as:

```
with logger.log_to_file('myfile.log'):  
    # your code here
```

log_to_list (*args, **kws)

Context manager to temporarily log messages to a list.

Parameters

filename : str

The file to log messages to.

filter_level : str

If set, any log messages less important than `filter_level` will not be output to the file. Note that this is in addition to the top-level filtering for the logger, so if the logger has level 'INFO', then setting `filter_level` to INFO or DEBUG will have no effect, since these messages are already filtered out.

filter_origin : str

If set, only log messages with an origin starting with `filter_origin` will be output to the file.

Notes

Using this context manager does not stop log messages from being output to standard output.

Examples

The context manager is used as:

```
with logger.log_to_list() as log_list:  
    # your code here
```

makeRecord (name, level, pathname, lineno, msg, args, exc_info, func=None, extra=None, sinfo=None)

setLevel (level)

Set the logging level of this logger.

warnings_logging_enabled ()

LoggingError

exception `astropy.logger.LoggingError`

This exception is for various errors that occur in the astropy logger, typically when activating or deactivating logger-related features.

PYTHON WARNINGS SYSTEM

Astropy uses the Python `warnings` module to issue warning messages. The details of using the warnings module are general to Python, and apply to any Python software that uses this system. The user can suppress the warnings using the python command line argument `-W"ignore"` when starting an interactive python session. For example:

```
$ python -W"ignore"
```

The user may also use the command line argument when running a python script as follows:

```
$ python -W"ignore" myscript.py
```

It is also possible to suppress warnings from within a python script. For instance, the warnings issued from a single call to the `astropy.io.fits.writeto` function may be suppressed from within a Python script as follows:

```
>>> import warnings
>>> from astropy.io import fits
>>> warnings.filterwarnings('ignore', category=UserWarning, append=True)
>>> fits.writeto(filename, data, clobber=True)
```

Astropy includes its own warning class, `AstropyUserWarning`, on which all warnings from Astropy are based. So one can also ignore warnings from Astropy (while still allowing through warnings from other libraries like Numpy) by using something like:

```
>>> warnings.filterwarnings('ignore', category=AstropyUserWarning)
```

However, warning filters may also be modified just within a certain context using the `catch_warnings` context manager:

```
>>> from warnings import catch_warnings
>>> with catch_warnings():
...     warnings.filterwarnings('ignore', AstropyUserWarning)
...     fits.writeto(filename, data, clobber=True)
```

Astropy also issues warnings when deprecated API features are used. If you wish to *squelch* deprecation warnings, you can start Python with `-Wi::Deprecation`. This sets all deprecation warnings to ignored. There is also an Astropy-specific `AstropyDeprecationWarning` which can be used to disable deprecation warnings from Astropy only.

See <http://docs.python.org/using/cmdline.html#cmdoption-unittest-discover-W> for more information on the `-W` argument.

ASTROPY CORE PACKAGE UTILITIES (ASTROPY.UTILS)

26.1 Introduction

The `astropy.utils` package contains general-purpose utility functions and classes. Examples include data structures, tools for downloading and caching from URLs, and version intercompatibility functions.

This functionality is not astronomy-specific, but is intended primarily for use by Astropy developers. It is all safe for users to use, but the functions and classes are typically more complicated or specific to a particular need of Astropy.

Because of the mostly standalone and grab-bag nature of these utilities, they are generally best understood through their docstrings, and hence this documentation does not have detailed sections like the other packages.

Note: The `astropy.utils.compat` subpackage is not included in this documentation. It contains utility modules for compatibility with older/newer versions of python, as well as including some bugfixes for the stdlib that are important for Astropy. It is recommended that developers at least glance over the source code for this subpackage, but it cannot be reliably included here because of the large amount of version-specific code it contains.

26.2 Reference/API

26.2.1 `astropy.utils.misc` Module

A “grab bag” of relatively small general-purpose utilities that don’t have a clear module/package to live in.

Functions

<code>find_current_module([depth, finddiff])</code>	Determines the module/package from which this function is called.
<code>isiterable(obj)</code>	Returns <code>True</code> if the given object is iterable.
<code>deprecated(since[, message, name, ...])</code>	Used to mark a function or class as deprecated.
<code>deprecated_attribute(name, since[, message, ...])</code>	Used to mark a public attribute as deprecated.
<code>silence(*args, **kwds)</code>	A context manager that silences <code>sys.stdout</code> and <code>sys.stderr</code> .
<code>format_exception(msg, *args, **kwargs)</code>	Given an exception message string, uses new-style formatting arguments.
<code>find_api_page(obj[, version, openinbrowser, ...])</code>	Determines the URL of the API page for the specified object, and optionally opens it in a browser.
<code>is_path_hidden(filepath)</code>	Determines if a given file or directory is hidden.
<code>walk_skip_hidden(top[, onerror, followlinks])</code>	A wrapper for <code>os.walk</code> that skips hidden files and directories.
<code>indent(s[, shift, width])</code>	Indent a block of text.

find_current_module

`astropy.utils.misc.find_current_module` (*depth=1, finddiff=False*)

Determines the module/package from which this function is called.

This function has two modes, determined by the `finddiff` option. It will either simply go the requested number of frames up the call stack (if `finddiff` is `False`), or it will go up the call stack until it reaches a module that is *not* in a specified set.

Parameters

depth : int

Specifies how far back to go in the call stack (0-indexed, so that passing in 0 gives back `astropy.utils.misc`).

finddiff : bool or list

If `False`, the returned `mod` will just be `depth` frames up from the current frame. Otherwise, the function will start at a frame `depth` up from current, and continue up the call stack to the first module that is *different* from those in the provided list. In this case, `finddiff` can be a list of modules or modules names. Alternatively, it can be `True`, which will use the module `depth` call stack frames up as the module the returned module must be different from.

Returns

mod : module or None

The module object or `None` if the package cannot be found. The name of the module is available as the `__name__` attribute of the returned object (if it isn't `None`).

Raises

ValueError

If `finddiff` is a list with an invalid entry.

Examples

The examples below assume that there are two modules in a package named `pkg.mod1.py`:

```
def find1():
    from astropy.utils import find_current_module
    print find_current_module(1).__name__
def find2():
    from astropy.utils import find_current_module
    cmod = find_current_module(2)
    if cmod is None:
        print 'None'
    else:
        print cmod.__name__
def find_diff():
    from astropy.utils import find_current_module
    print find_current_module(0, True).__name__
```

`mod2.py`:

```
def find():
    from .mod1 import find2
    find2()
```

With these modules in place, the following occurs:

```
>>> from pkg import mod1, mod2
>>> from astropy.utils import find_current_module
>>> mod1.find1()
pkg.mod1
>>> mod1.find2()
None
>>> mod2.find()
pkg.mod2
>>> find_current_module(0)
<module 'astropy.utils.misc' from 'astropy/utils/misc.py'>
>>> mod1.find_diff()
pkg.mod1
```

isiterable

`astropy.utils.misc.isiterable(obj)`
Returns `True` if the given object is iterable.

deprecated

`astropy.utils.misc.deprecated(since, message=u'', name=u'', alternative=u'', pending=False, obj_type=None)`

Used to mark a function or class as deprecated.

To mark an attribute as deprecated, use `deprecated_attribute`.

Parameters

since : str

The release at which this API became deprecated. This is required.

message : str, optional

Override the default deprecation message. The format specifier `func` may be used for the name of the function, and `alternative` may be used in the deprecation message to insert the name of an alternative to the deprecated function. `obj_type` may be used to insert a friendly name for the type of object being deprecated.

name : str, optional

The name of the deprecated function or class; if not provided the name is automatically determined from the passed in function or class, though this is useful in the case of re-named functions, where the new function is just assigned to the name of the deprecated function. For example:

```
def new_function():
    ...
oldFunction = new_function
```

alternative : str, optional

An alternative function or class name that the user may use in place of the deprecated object. The deprecation warning will tell the user about this alternative if provided.

pending : bool, optional

If True, uses a `AstropyPendingDeprecationWarning` instead of a `AstropyDeprecationWarning`.

obj_type : str, optional

The type of this object, if the automatically determined one needs to be overridden.

deprecated_attribute

`astropy.utils.misc.deprecated_attribute` (*name*, *since*, *message=None*, *alternative=None*, *pending=False*)

Used to mark a public attribute as deprecated. This creates a property that will warn when the given attribute name is accessed. To prevent the warning (i.e. for internal code), use the private name for the attribute by prepending an underscore (i.e. `self._name`).

Parameters

name : str

The name of the deprecated attribute.

since : str

The release at which this API became deprecated. This is required.

message : str, optional

Override the default deprecation message. The format specifier `name` may be used for the name of the attribute, and `alternative` may be used in the deprecation message to insert the name of an alternative to the deprecated function.

alternative : str, optional

An alternative attribute that the user may use in place of the deprecated attribute. The deprecation warning will tell the user about this alternative if provided.

pending : bool, optional

If True, uses a `AstropyPendingDeprecationWarning` instead of a `AstropyDeprecationWarning`.

Examples

```
class MyClass:
    # Mark the old_name as deprecated
    old_name = misc.deprecated_attribute('old_name', '0.1')

    def method(self):
        self._old_name = 42
```

silence

`astropy.utils.misc.silence` (**args*, ***kwds*)
A context manager that silences `sys.stdout` and `sys.stderr`.

format_exception

`astropy.utils.misc.format_exception(msg, *args, **kwargs)`

Given an exception message string, uses new-style formatting arguments `{filename}`, `{lineno}`, `{func}` and/or `{text}` to fill in information about the exception that occurred. For example:

```
try:
    1/0
except:
```

```
    raise ZeroDivisionError(
```

```
        format_except('A divide by zero occurred in {filename} at '
                       'line {lineno} of function {func}.')
```

Any additional positional or keyword arguments passed to this function are also used to format the message.

Note: This uses `sys.exc_info` to gather up the information needed to fill in the formatting arguments. Python 2.x and 3.x have slightly different behavior regarding `sys.exc_info` (the latter will not carry it outside a handled exception), so it's not wise to use this outside of an `except` clause - if it is, this will substitute '<unkown>' for the 4 formatting arguments.

find_api_page

`astropy.utils.misc.find_api_page(obj, version=None, openinbrowser=True, timeout=None)`

Determines the URL of the API page for the specified object, and optionally open that page in a web browser.

Note: You must be connected to the internet for this to function even if `openinbrowser` is `False`, unless you provide a local version of the documentation to `version` (e.g., `file:///path/to/docs`).

Parameters

obj

The object to open the docs for or its fully-qualified name (as a str).

version : str

The doc version - either a version number like '0.1', 'dev' for the development/latest docs, or a URL to point to a specific location that should be the *base* of the documentation. Defaults to latest if you are on aren't on a release, otherwise, the version you are on.

openinbrowser : bool

If `True`, the `webbrowser` package will be used to open the doc page in a new web browser window.

timeout : number, optional

The number of seconds to wait before timing-out the query to the astropy documentation. If not given, the default python `stdlib` timeout will be used.

Returns

url : str

The loaded URL

Raises**ValueError**

If the documentation can't be found

is_path_hidden

`astropy.utils.misc.is_path_hidden(filepath)`

Determines if a given file or directory is hidden.

Parameters

filepath : str

The path to a file or directory

Returns

hidden : bool

Returns `True` if the file is hidden

walk_skip_hidden

`astropy.utils.misc.walk_skip_hidden(top, onerror=None, followlinks=False)`

A wrapper for `os.walk` that skips hidden files and directories.

This function does not have the parameter `topdown` from `os.walk`: the directories must always be recursed top-down when using this function.

See also:**`os.walk`**

For a description of the parameters

indent

`astropy.utils.misc.indent(s, shift=1, width=4)`

Indent a block of text. The indentation is applied to each line.

Classes

<code>lazyproperty(fget[, fset, fdel, doc])</code>	Works similarly to <code>property()</code> , but computes the value only once.
<code>NumpyRNGContext(seed)</code>	A context manager (for use with the <code>with</code> statement) that will seed the numpy RNG.
<code>JsonCustomEncoder([skipkeys, ensure_ascii, ...])</code>	Support for data types that JSON default encoder does not do.
<code>InheritDocstrings(name, bases, dct)</code>	This metaclass makes methods of a class automatically have their docstrings.

lazyproperty

class `astropy.utils.misc.lazyproperty(fget, fset=None, fdel=None, doc=None)`

Bases: `object`

Works similarly to `property()`, but computes the value only once.

This essentially memoizes the value of the property by storing the result of its computation in the `__dict__` of the object instance. This is useful for computing the value of some property that should otherwise be invariant. For example:

```
>>> class LazyTest(object):
...     @lazyproperty
...     def complicated_property(self):
...         print('Computing the value for complicated_property...')
...         return 42
...
>>> lt = LazyTest()
>>> lt.complcated_property
Computing the value for complicated_property...
42
>>> lt.complcated_property
42
```

If a setter for this property is defined, it will still be possible to manually update the value of the property, if that capability is desired.

Adapted from the recipe at <http://code.activestate.com/recipes/363602-lazy-property-evaluation>

Methods Summary

```
deleter(fdel)
getter(fget)
setter(fset)
```

Methods Documentation

deleter (*fdel*)

getter (*fget*)

setter (*fset*)

NumpyRNGContext

class `astropy.utils.misc.NumpyRNGContext` (*seed*)
Bases: `object`

A context manager (for use with the `with` statement) that will seed the numpy random number generator (RNG) to a specific value, and then restore the RNG state back to whatever it was before.

This is primarily intended for use in the astropy testing suit, but it may be useful in ensuring reproducibility of Monte Carlo simulations in a science context.

Parameters

seed : int

The value to use to seed the numpy RNG

Examples

A typical use case might be:

```
with NumpyRNGContext(<some seed value you pick>):
    from numpy import random

    randarr = random.randn(100)
    ... run your test using `randarr` ...

#Any code using numpy.random at this indent level will act just as it
#would have if it had been before the with statement - e.g. whatever
#the default seed is.
```

JsonCustomEncoder

```
class astropy.utils.misc.JsonCustomEncoder(skipkeys=False, ensure_ascii=True,
                                           check_circular=True, allow_nan=True,
                                           sort_keys=False, indent=None, separa-
                                           tors=None, encoding='utf-8', default=None)
```

Bases: `json.encoder.JSONEncoder`

Support for data types that JSON default encoder does not do.

This includes:

- Numpy array or number
- Complex number
- Set
- Bytes (Python 3)

Examples

```
>>> import json
>>> import numpy as np
>>> from astropy.utils.misc import JsonCustomEncoder
>>> json.dumps(np.arange(3), cls=JsonCustomEncoder)
'[0, 1, 2]'
```

Constructor for JSONEncoder, with sensible defaults.

If `skipkeys` is `False`, then it is a `TypeError` to attempt encoding of keys that are not `str`, `int`, `long`, `float` or `None`. If `skipkeys` is `True`, such items are simply skipped.

If `ensure_ascii` is `True`, the output is guaranteed to be `str` objects with all incoming unicode characters escaped. If `ensure_ascii` is `false`, the output will be unicode object.

If `check_circular` is `True`, then lists, dicts, and custom encoded objects will be checked for circular references during encoding to prevent an infinite recursion (which would cause an `OverflowError`). Otherwise, no such check takes place.

If `allow_nan` is `True`, then `NaN`, `Infinity`, and `-Infinity` will be encoded as such. This behavior is not JSON specification compliant, but is consistent with most JavaScript based encoders and decoders. Otherwise, it will be a `ValueError` to encode such floats.

If `sort_keys` is `True`, then the output of dictionaries will be sorted by key; this is useful for regression tests to ensure that JSON serializations can be compared on a day-to-day basis.

If `indent` is a non-negative integer, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0 will only insert newlines. None is the most compact representation.

If specified, `separators` should be a (`item_separator`, `key_separator`) tuple. The default is (`' , '`, `' : '`). To get the most compact JSON representation you should specify (`' , '`, `' : '`) to eliminate whitespace.

If specified, `default` is a function that gets called for objects that can't otherwise be serialized. It should return a JSON encodable version of the object or raise a `TypeError`.

If encoding is not `None`, then all input strings will be transformed into unicode using that encoding prior to JSON-encoding. The default is UTF-8.

Methods Summary

`default(obj)`

Methods Documentation

default (*obj*)

InheritDocstrings

class `astropy.utils.misc.InheritDocstrings` (*name, bases, dct*)

Bases: `type`

This metaclass makes methods of a class automatically have their docstrings filled in from the methods they override in the base class.

If the class uses multiple inheritance, the docstring will be chosen from the first class in the bases list, in the same way as methods are normally resolved in Python. If this results in selecting the wrong docstring, the docstring will need to be explicitly included on the method.

For example:

```
>>> from astropy.utils.misc import InheritDocstrings
>>> from astropy.extern import six
>>> @six.add_metaclass(InheritDocstrings)
... class A(object):
...     def wiggle(self):
...         "Wiggle the thingamajig"
...     pass
>>> class B(A):
...     def wiggle(self):
...         pass
>>> B.wiggle.__doc__
u'Wiggle the thingamajig'
```

26.2.2 astropy.utils.exceptions Module

This module contains errors/exceptions and warnings of general use for astropy. Exceptions that are specific to a given subpackage should *not* be here, but rather in the particular subpackage.

Classes

<code>AstropyBackwardsIncompatibleChangeWarning</code>	A warning class indicating a change in astropy that is incompatible with previous versions.
<code>AstropyDeprecationWarning</code>	A warning class to indicate a deprecated feature.
<code>AstropyPendingDeprecationWarning</code>	A warning class to indicate a soon-to-be deprecated feature.
<code>AstropyUserWarning</code>	The primary warning class for Astropy.
<code>AstropyWarning</code>	The base warning class from which all Astropy warnings should inherit.

AstropyBackwardsIncompatibleChangeWarning

exception `astropy.utils.exceptions.AstropyBackwardsIncompatibleChangeWarning`

A warning class indicating a change in astropy that is incompatible with previous versions.

The suggested procedure is to issue this warning for the version in which the change occurs, and remove it for all following versions.

AstropyDeprecationWarning

exception `astropy.utils.exceptions.AstropyDeprecationWarning`

A warning class to indicate a deprecated feature.

AstropyPendingDeprecationWarning

exception `astropy.utils.exceptions.AstropyPendingDeprecationWarning`

A warning class to indicate a soon-to-be deprecated feature.

AstropyUserWarning

exception `astropy.utils.exceptions.AstropyUserWarning`

The primary warning class for Astropy.

Use this if you do not need a specific sub-class.

AstropyWarning

exception `astropy.utils.exceptions.AstropyWarning`

The base warning class from which all Astropy warnings should inherit.

Any warning inheriting from this class is handled by the Astropy logger.

26.2.3 astropy.utils.collections Module

A module containing specialized collection classes.

Classes

`HomogeneousList(types[, values])` A subclass of list that contains only elements of a given type or types.

HomogeneousList

class `astropy.utils.collections.HomogeneousList` (*types*, *values=[]*)

Bases: `list`

A subclass of list that contains only elements of a given type or types. If an item that is not of the specified type is added to the list, a `TypeError` is raised.

Parameters

types : sequence of types

The types to accept.

values : sequence, optional

An initial set of values.

Methods Summary

```
append(x)
extend(x)
insert(i, x)
```

Methods Documentation

append (*x*)

extend (*x*)

insert (*i*, *x*)

26.2.4 astropy.utils.console Module

Utilities for console input and output.

Functions

<code>isatty(file)</code>	Returns <code>True</code> if <code>file</code> is a tty.
<code>color_print(*args, **kwargs)</code>	Prints colors and styles to the terminal uses ANSI escape sequences.
<code>human_time(seconds)</code>	Returns a human-friendly time string that is always exactly 6 characters long.
<code>human_file_size(size)</code>	Returns a human-friendly string representing a file size that is 2-4 characters long.
<code>print_code_line(line[, col, file, tabwidth, ...])</code>	Prints a line of source code, highlighting a particular character position in the line.
<code>terminal_size([file])</code>	Returns a tuple (height, width) containing the height and width of the terminal.

isatty

`astropy.utils.console.isatty(file)`

Returns `True` if `file` is a tty.

Most built-in Python file-like objects have an `isatty` member, but some user-defined types may not, so this assumes those are not ttys.

color_print

`astropy.utils.console.color_print(*args, **kwargs)`

Prints colors and styles to the terminal uses ANSI escape sequences.

```
color_print('This is the color ', 'default', 'GREEN', 'green')
```

Parameters

positional args : str

The positional arguments come in pairs (*msg*, *color*), where *msg* is the string to display and *color* is the color to display it in.

color is an ANSI terminal color name. Must be one of: black, red, green, brown, blue, magenta, cyan, lightgrey, default, darkgrey, lightred, lightgreen, yellow, lightblue, lightmagenta, lightcyan, white, or "" (the empty string).

file : writeable file-like object, optional

Where to write to. Defaults to `sys.stdout`. If `file` is not a tty (as determined by calling its `isatty` member, if one exists), no coloring will be included.

end : str, optional

The ending of the message. Defaults to `\n`. The end will be printed after resetting any color or font state.

human_time

`astropy.utils.console.human_time(seconds)`

Returns a human-friendly time string that is always exactly 6 characters long.

Depending on the number of seconds given, can be one of:

```
1w 3d
2d 4h
1h 5m
1m 4s
15s
```

Will be in color if console coloring is turned on.

Parameters

seconds : int

The number of seconds to represent

Returns

time : str

A human-friendly representation of the given number of seconds that is always exactly 6 characters.

human_file_size

`astropy.utils.console.human_file_size` (*size*)

Returns a human-friendly string representing a file size that is 2-4 characters long.

For example, depending on the number of bytes given, can be one of:

```
256b
64k
1.1G
```

Parameters

size : int

The size of the file (in bytes)

Returns

size : str

A human-friendly representation of the size of the file

print_code_line

`astropy.utils.console.print_code_line` (*line*, *col=None*, *file=None*, *tabwidth=8*, *width=70*)

Prints a line of source code, highlighting a particular character position in the line. Useful for displaying the context of error messages.

If the line is more than `width` characters, the line is truncated accordingly and `'...'` characters are inserted at the front and/or end.

It looks like this:

```
there_is_a_syntax_error_here :
                               ^
```

Parameters

line : unicode

The line of code to display

col : int, optional

The character in the line to highlight. `col` must be less than `len(line)`.

file : writeable file-like object, optional

Where to write to. Defaults to `sys.stdout`.

tabwidth : int, optional

The number of spaces per tab (`'\t'`) character. Default is 8. All tabs will be converted to spaces to ensure that the caret lines up with the correct column.

width : int, optional

The width of the display, beyond which the line will be truncated. Defaults to 70 (this matches the default in the standard library's `textwrap` module).

terminal_size

`astropy.utils.console.terminal_size` (*file=None*)

Returns a tuple (height, width) containing the height and width of the terminal.

This function will look for the width in height in multiple areas before falling back on the width and height in astropy's configuration.

Classes

<code>ProgressBar(total_or_items[, file])</code>	A class to display a progress bar in the terminal.
<code>Spinner(msg[, color, file, step, chars])</code>	A class to display a spinner in the terminal.
<code>ProgressBarOrSpinner(total, msg[, color, file])</code>	A class that displays either a <code>ProgressBar</code> or <code>Spinner</code> depending on wh

ProgressBar

class `astropy.utils.console.ProgressBar` (*total_or_items, file=None*)

Bases: `astropy.extern.six.Iterator`

A class to display a progress bar in the terminal.

It is designed to be used either with the `with` statement:

```
with ProgressBar(len(items)) as bar:
    for item in enumerate(items):
        bar.update()
```

or as a generator:

```
for item in ProgressBar(items):
    item.process()
```

Parameters

total_or_items : int or sequence

If an int, the number of increments in the process being tracked. If a sequence, the items to iterate over.

file : writable file-like object, optional

The file to write the progress bar to. Defaults to `sys.stdout`. If `file` is not a tty (as determined by calling its `isatty` member, if any, or special case hacks to detect the IPython console), the progress bar will be completely silent.

Methods Summary

<code>iterate(*args, **kwargs)</code>	Deprecated since version 0.3.
<code>map(function, items[, multiprocessing, file])</code>	Does a <code>map</code> operation while displaying a progress bar with percentage complete.
<code>update([value])</code>	Update the progress bar to the given value (out of the total given to the constructor).

Methods Documentation

classmethod `iterate` (*args, **kwargs)

Deprecated since version 0.3: The `iterate` method is deprecated and may be removed in a future version. Use `ProgressBar` instead.

Iterate over a sequence while indicating progress with a progress bar in the terminal.

```
for item in ProgressBar.iterate(items):
    pass
```

Parameters

items : sequence

A sequence of items to iterate over

file : writeable file-like object, optional

The file to write the progress bar to. Defaults to `sys.stdout`. If `file` is not a tty (as determined by calling its `isatty` member, if any), the scrollbar will be completely silent.

Returns

generator :

A generator over `items`.

classmethod `map` (function, items, multiprocessing=False, file=None)

Does a `map` operation while displaying a progress bar with percentage complete.

```
def work(i):
    print(i)
```

```
ProgressBar.map(work, range(50))
```

Parameters

function : function

Function to call for each step

items : sequence

Sequence where each element is a tuple of arguments to pass to *function*.

multiprocess : bool, optional

If `True`, use the `multiprocessing` module to distribute each task to a different processor core.

file : writeable file-like object, optional

The file to write the progress bar to. Defaults to `sys.stdout`. If `file` is not a tty (as determined by calling its `isatty` member, if any), the scrollbar will be completely silent.

update (value=None)

Update the progress bar to the given value (out of the total given to the constructor).

Spinner

class `astropy.utils.console.Spinner` (*msg*, *color=u'default'*, *file=None*, *step=1*, *chars=None*)
Bases: `object`

A class to display a spinner in the terminal.

It is designed to be used with the `with` statement:

```
with Spinner("Reticulating splines", "green") as s:
    for item in enumerate(items):
        s.next()
```

Parameters

msg : str

The message to print

color : str, optional

An ANSI terminal color name. Must be one of: black, red, green, brown, blue, magenta, cyan, lightgrey, default, darkgrey, lightred, lightgreen, yellow, lightblue, lightmagenta, lightcyan, white.

file : writeable file-like object, optional

The file to write the spinner to. Defaults to `sys.stdout`. If `file` is not a tty (as determined by calling its `isatty` member, if any, or special case hacks to detect the IPython console), the spinner will be completely silent.

step : int, optional

Only update the spinner every *step* steps

chars : str, optional

The character sequence to use for the spinner

ProgressBarOrSpinner

class `astropy.utils.console.ProgressBarOrSpinner` (*total*, *msg*, *color=u'default'*, *file=None*)
Bases: `object`

A class that displays either a `ProgressBar` or `Spinner` depending on whether the total size of the operation is known or not.

It is designed to be used with the `with` statement:

```
if file.has_length():
    length = file.get_length()
else:
    length = None
bytes_read = 0
with ProgressBarOrSpinner(length) as bar:
    while file.read(blocksize):
        bytes_read += blocksize
        bar.update(bytes_read)
```

Parameters

total : int or None

If an int, the number of increments in the process being tracked and a `ProgressBar` is displayed. If `None`, a `Spinner` is displayed.

msg : str

The message to display above the `ProgressBar` or alongside the `Spinner`.

color : str, optional

The color of `msg`, if any. Must be an ANSI terminal color name. Must be one of: black, red, green, brown, blue, magenta, cyan, lightgrey, default, darkgrey, lightred, lightgreen, yellow, lightblue, lightmagenta, lightcyan, white.

file : writable file-like object, optional

The file to write the to. Defaults to `sys.stdout`. If `file` is not a tty (as determined by calling its `isatty` member, if any), only `msg` will be displayed: the `ProgressBar` or `Spinner` will be silent.

Methods Summary

`update(value)` Update the progress bar to the given value (out of the total given to the constructor).

Methods Documentation

update (*value*)

Update the progress bar to the given value (out of the total given to the constructor).

26.2.5 astropy.utils.timer Module

General purpose timer related functions.

Functions

`timefunc([num_tries, verbose])` Decorator to time a function or method.

timefunc

`astropy.utils.timer.timefunc` (*num_tries=1, verbose=True*)

Decorator to time a function or method.

Parameters

num_tries : int, optional

Number of calls to make. Timer will take the average run time.

verbose : bool, optional

Extra log information.

Returns

tt : float

Average run time in seconds.

result

Output(s) from the function.

Examples

To add timer to time `numpy.log` for 100 times with verbose output:

```
import numpy as np
from astropy.utils.timer import timefunc
```

```
@timefunc(100)
def timed_log(x):
    return np.log(x)
```

To run the decorated function above:

```
>>> t, y = timed_log(100)
INFO: timed_log took 9.29832458496e-06 s on AVERAGE for 100 call(s). [...]
>>> t
9.298324584960938e-06
>>> y
4.6051701859880918
```

Classes

`RunTimePredictor(func, *args, **kwargs)` Class to predict run time.

RunTimePredictor

class `astropy.utils.timer.RunTimePredictor` (*func, *args, **kwargs*)

Bases: `object`

Class to predict run time.

Note: Only predict for single varying numeric input parameter.

Parameters

func : function

Function to time.

args : tuple

Fixed positional argument(s) for the function.

kwargs : dict

Fixed keyword argument(s) for the function.

Examples

```
>>> from astropy.utils.timer import RunTimePredictor
```

Set up a predictor for 10^x :

```
>>> p = RunTimePredictor(pow, 10)
```

Give it baseline data to use for prediction and get the function output values:

```
>>> p.time_func(range(10, 1000, 200))
>>> for input, result in sorted(p.results.items()):
...     print("pow(10, {0})\n{1}".format(input, result))
pow(10, 10)
10000000000
pow(10, 210)
10000000000...
pow(10, 410)
10000000000...
pow(10, 610)
10000000000...
pow(10, 810)
10000000000...
```

Fit a straight line assuming \arg^1 relationship (coefficients are returned):

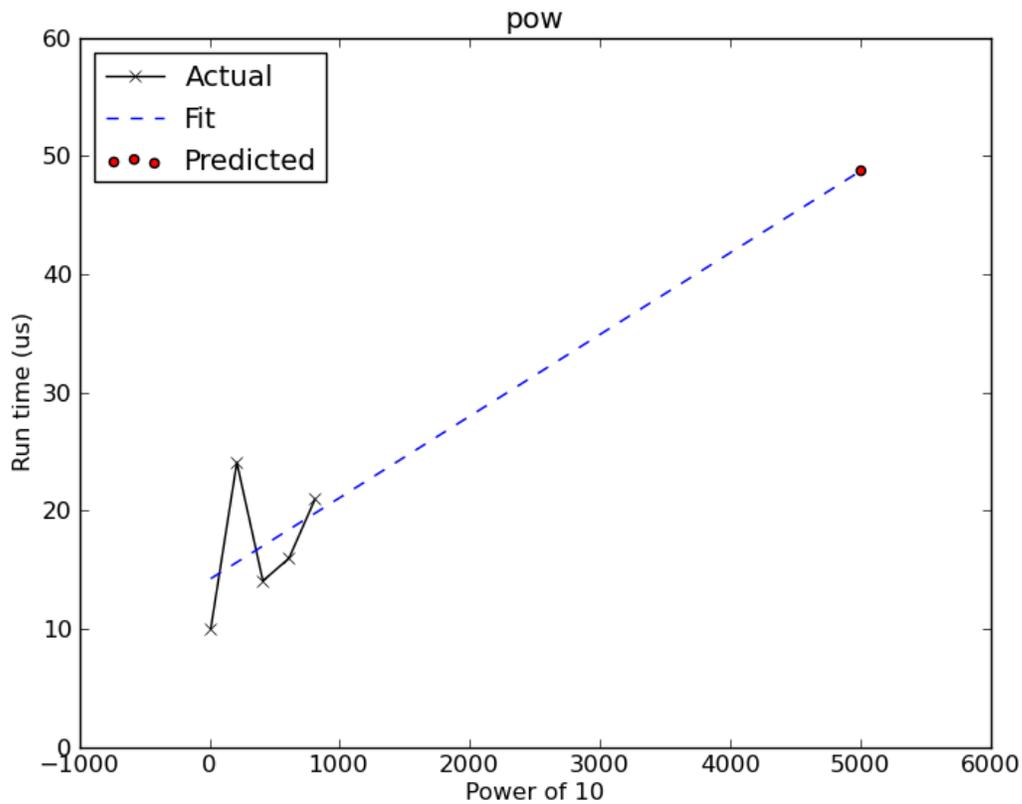
```
>>> p.do_fit()
array([1.16777420e-05, 1.00135803e-08])
```

Predict run time for 10^{5000} :

```
>>> p.predict_time(5000)
6.174564361572262e-05
```

Plot the prediction:

```
>>> p.plot(xlabeltext='Power of 10')
```



When the changing argument is not the last, e.g., x^2 , something like this might work:

```
>>> p = RunTimePredictor(lambda x: pow(x, 2))
>>> p.time_func([2, 3, 5])
>>> sorted(p.results.items())
[(2, 4), (3, 9), (5, 25)]
```

Attributes Summary

`results` Function outputs from `time_func`.

Methods Summary

<code>do_fit([model, fitter, power, min_datapoints])</code>	Fit a function to the lists of arguments and their respective run time in the cache.
<code>plot([xscale, yscale, xlabeltext, save_as])</code>	Plot prediction.
<code>predict_time(arg)</code>	Predict run time for given argument.
<code>time_func(arglist)</code>	Time the partial function for a list of single args and store run time in a cache.

Attributes Documentation

results

Function outputs from `time_func`.

A dictionary mapping input arguments (fixed arguments are not included) to their respective output values.

Methods Documentation

do_fit (*model=None, fitter=None, power=1, min_datapoints=3*)

Fit a function to the lists of arguments and their respective run time in the cache.

By default, this does a linear least-square fitting to a straight line on run time w.r.t. argument values raised to the given power, and returns the optimal intercept and slope.

Parameters

model : `astropy.modeling.Model`

Model for the expected trend of run time (Y-axis) w.r.t. $\text{arg}^{\text{power}}$ (X-axis). If `None`, will use `Polynomial1D` with `degree=1`.

fitter : `astropy.modeling.fitting.Fitter`

Fitter for the given model to extract optimal coefficient values. If `None`, will use `LinearLSQFitter`.

power : int, optional

Power of values to fit.

min_datapoints : int, optional

Minimum number of data points required for fitting. They can be built up with `time_func`.

Returns

a : array_like

Fitted `FittableModel` parameters.

Raises

AssertionError

Insufficient data points for fitting.

ModelError

Invalid model or fitter.

plot (*xscale=u'linear', yscale=u'linear', xlabeltext=u'args', save_as=u''*)

Plot prediction.

Note: Uses `matplotlib`.

Parameters

xscale, yscale : {'linear', 'log', 'symlog'}

Scaling for `matplotlib.axes.Axes`.

xlabeltext : str, optional

Text for X-label.

save_as : str, optional

Save plot as given filename.

Raises**AssertionError**

Insufficient data for plotting.

predict_time (*arg*)

Predict run time for given argument. If prediction is already cached, cached value is returned.

Parameters

arg : number

Input argument to predict run time for.

Returns

t_est : float

Estimated run time for given argument.

Raises**AssertionError**

No fitted data for prediction.

time_func (*arglist*)

Time the partial function for a list of single args and store run time in a cache. This forms a baseline for the prediction.

This also stores function outputs in `results`.

Parameters

arglist : list of numbers

List of input arguments to time.

26.2.6 astropy.utils.state Module

A simple class to manage a piece of global science state. See *Adding new configuration items* for more details.

Classes

<code>ScienceState()</code>	Science state subclasses are used to manage global items that can affect science results.
<code>ScienceStateAlias</code>	This is a backward compatibility layer for configuration items that moved to <code>ScienceState</code> classes in astropy.

ScienceState

```
class astropy.utils.state.ScienceState
```

```
    Bases: object
```

Science state subclasses are used to manage global items that can affect science results. Subclasses will generally override `validate` to convert from any of the acceptable inputs (such as strings) to the appropriate internal objects, and set an initial value to the `_value` member so it has a default.

Examples

```
class MyState(ScienceState):
    @classmethod
```

```
def validate(cls, value):
    if value not in ('A', 'B', 'C'):
        raise ValueError("Must be one of A, B, C")
    return value
```

Methods Summary

<code>get()</code>	Get the current science state value.
<code>set(value)</code>	Set the current science state value.
<code>validate(value)</code>	Validate the value and convert it to its native type, if necessary.

Methods Documentation

classmethod `get()`

Get the current science state value.

classmethod `set(value)`

Set the current science state value.

classmethod `validate(value)`

Validate the value and convert it to its native type, if necessary.

`state.ScienceStateAlias`

`state.ScienceStateAlias`

This is a backward compatibility layer for configuration items that moved to ScienceState classes in astropy 0.4.

Parameters

since : str

The version in which the configuration item was converted into science state.

python_name : str

The old name of the Python variable for the configuration item.

config_name : str

The old name of the configuration item in the configuration file.

science_state : ScienceState subclass

The science state class that now manages this information.

cfgtype : str or `None`, optional

A type specifier like those used as the *values* of a particular key in a `configspec` file of `configobj`. If `None`, the type will be inferred from the default value.

module : str, optional

The module containing the old configuration item.

26.2.7 File Downloads

26.2.8 `astropy.utils.data` Module

This module contains helper functions for accessing, downloading, and caching data files.

Functions

<code>get_readable_fileobj(*args, **kwds)</code>	Given a filename or a readable file-like object, return a context manager that yields a readable file-like object.
<code>get_file_contents(name_or_obj[, encoding, cache])</code>	Retrieves the contents of a filename or file-like object.
<code>get_pkg_data_fileobj(data_name[, encoding, ...])</code>	Retrieves a data file from the standard locations for the package and provides a readable file-like object.
<code>get_pkg_data_filename(data_name[, ...])</code>	Retrieves a data file from the standard locations for the package and provides the path to the file.
<code>get_pkg_data_contents(data_name[, encoding, ...])</code>	Retrieves a data file from the standard locations and returns its contents.
<code>get_pkg_data_fileobjs(datadir[, pattern, ...])</code>	Returns readable file objects for all of the data files in a given directory.
<code>get_pkg_data_filenames(datadir[, pattern])</code>	Returns the path of all of the data files in a given directory that match a given pattern.
<code>compute_hash(localfn)</code>	Computes the MD5 hash for a file.
<code>clear_download_cache([hashorurl])</code>	Clears the data file cache by deleting the local file(s).
<code>get_free_space_in_dir(path)</code>	Given a path to a directory, returns the amount of free space (in bytes) on the disk.
<code>check_free_space_in_dir(path, size)</code>	Determines if a given directory has enough space to hold a file of a given size.
<code>download_file(remote_url[, cache, ...])</code>	Accepts a URL, downloads and optionally caches the result returning the local path.
<code>download_files_in_parallel(urls[, cache, ...])</code>	Downloads multiple files in parallel from the given URLs.

`get_readable_fileobj`

`astropy.utils.data.get_readable_fileobj(*args, **kwds)`

Given a filename or a readable file-like object, return a context manager that yields a readable file-like object.

This supports passing filenames, URLs, and readable file-like objects, any of which can be compressed in gzip or bzip2.

Parameters

name_or_obj : str or file-like object

The filename of the file to access (if given as a string), or the file-like object to access.

If a file-like object, it must be opened in binary mode.

encoding : str, optional

When `None` (default), returns a file-like object with a `read` method that on Python 2.x returns `bytes` objects and on Python 3.x returns `str (unicode)` objects, using `locale.getpreferredencoding` as an encoding. This matches the default behavior of the built-in `open` when no `mode` argument is provided.

When `'binary'`, returns a file-like object where its `read` method returns `bytes` objects.

When another string, it is the name of an encoding, and the file-like object's `read` method will return `str (unicode)` objects, decoded from binary using the given encoding.

cache : bool, optional

Whether to cache the contents of remote URLs.

show_progress : bool, optional

Whether to display a progress bar if the file is downloaded from a remote server. Default is `True`.

remote_timeout : float

Timeout for remote requests in seconds (default is the configurable `REMOTE_TIMEOUT`, which is 3s by default)

Returns

file : readable file-like object

Notes

This function is a context manager, and should be used for example as:

```
with get_readable_fileobj('file.dat') as f:
    contents = f.read()
```

get_file_contents

`astropy.utils.data.get_file_contents` (*name_or_obj*, *encoding=None*, *cache=False*)

Retrieves the contents of a filename or file-like object.

See the `get_readable_fileobj` docstring for details on parameters.

Returns

content

The content of the file (as requested by *encoding*).

get_pkg_data_fileobj

`astropy.utils.data.get_pkg_data_fileobj` (*data_name*, *encoding=None*, *cache=True*)

Retrieves a data file from the standard locations for the package and provides the file as a file-like object that reads bytes.

Parameters

data_name : str

Name/location of the desired data file. One of the following:

- The name of a data file included in the source distribution. The path is relative to the module calling this function. For example, if calling from `astropy.pkname`, use `'data/file.dat'` to get the file in `astropy/pkname/data/file.dat`. Double-dots can be used to go up a level. In the same example, use `'../data/file.dat'` to get `astropy/data/file.dat`.
- If a matching local file does not exist, the Astropy data server will be queried for the file.
- A hash like that produced by `compute_hash` can be requested, prefixed by `'hash/'` e.g. `'hash/395dd6493cc584df1e78b474fb150840'`. The hash will first be searched for locally, and if not found, the Astropy data server will be queried.

encoding : str, optional

When `None` (default), returns a file-like object with a `read` method that on Python 2.x returns `bytes` objects and on Python 3.x returns `str (unicode)` objects, using `locale.getpreferredencoding` as an encoding. This matches the default behavior of the built-in `open` when no `mode` argument is provided.

When `'binary'`, returns a file-like object where its `read` method returns `bytes` objects.

When another string, it is the name of an encoding, and the file-like object's `read` method will return `str (unicode)` objects, decoded from binary using the given encoding.

cache : bool

If True, the file will be downloaded and saved locally or the already-cached local copy will be accessed. If False, the file-like object will directly access the resource (e.g. if a remote URL is accessed, an object like that from `urllib2.urlopen` on Python 2 or `urllib.request.urlopen` on Python 3 is returned).

Returns

fileobj : file-like

An object with the contents of the data file available via `read` function. Can be used as part of a `with` statement, automatically closing itself after the `with` block.

Raises

urllib2.URLError, urllib.error.URLError

If a remote file cannot be found.

IOError

If problems occur writing or reading a local file.

See also:

[`get_pkg_data_contents`](#)

returns the contents of a file or url as a bytes object

[`get_pkg_data_filename`](#)

returns a local name for a file containing the data

Examples

This will retrieve a data file and its contents for the `astropy.wcs` tests:

```
from astropy.utils.data import get_pkg_data_fileobj

with get_pkg_data_fileobj('data/3d_cd.hdr') as fobj:
    fcontents = fobj.read()
```

This would download a data file from the astropy data server because the `standards/vega.fits` file is not present in the source distribution. It will also save the file locally so the next time it is accessed it won't need to be downloaded.:

```
from astropy.utils.data import get_pkg_data_fileobj

with get_pkg_data_fileobj('standards/vega.fits') as fobj:
    fcontents = fobj.read()
```

This does the same thing but does *not* cache it locally:

```
with get_pkg_data_fileobj('standards/vega.fits', cache=False) as fobj:
    fcontents = fobj.read()
```

get_pkg_data_filename

```
astropy.utils.data.get_pkg_data_filename(data_name, show_progress=True,
                                         remote_timeout=None)
```

Retrieves a data file from the standard locations for the package and provides a local filename for the data.

This function is similar to `get_pkg_data_fileobj` but returns the file *name* instead of a readable file-like object. This means that this function must always cache remote files locally, unlike `get_pkg_data_fileobj`.

Parameters

data_name : str

Name/location of the desired data file. One of the following:

- The name of a data file included in the source distribution. The path is relative to the module calling this function. For example, if calling from `astropy.pkname`, use `'data/file.dat'` to get the file in `astropy/pkname/data/file.dat`. Double-dots can be used to go up a level. In the same example, use `'../data/file.dat'` to get `astropy/data/file.dat`.
- If a matching local file does not exist, the Astropy data server will be queried for the file.
- A hash like that produced by `compute_hash` can be requested, prefixed by `'hash/'` e.g. `'hash/395dd6493cc584df1e78b474fb150840'`. The hash will first be searched for locally, and if not found, the Astropy data server will be queried.

show_progress : bool, optional

Whether to display a progress bar if the file is downloaded from a remote server. Default is `True`.

remote_timeout : float

Timeout for the requests in seconds (default is the configurable `astropy.utils.data.Conf.remote_timeout`, which is 3s by default)

Returns

filename : str

A file path on the local file system corresponding to the data requested in `data_name`.

Raises

urllib2.URLError, urllib.error.URLError

If a remote file cannot be found.

IOError

If problems occur writing or reading a local file.

See also:

`get_pkg_data_contents`

returns the contents of a file or url as a bytes object

`get_pkg_data_fileobj`

returns a file-like object with the data

Examples

This will retrieve the contents of the data file for the `astropy.wcs` tests:

```
from astropy.utils.data import get_pkg_data_filename

fn = get_pkg_data_filename('data/3d_cd.hdr')
with open(fn) as f:
    fcontents = f.read()
```

This retrieves a data file by hash either locally or from the astropy data server:

```
from astropy.utils.data import get_pkg_data_filename

fn = get_pkg_data_filename('hash/da34a7b07ef153eede67387bf950bb32')
with open(fn) as f:
    fcontents = f.read()
```

`get_pkg_data_contents`

`astropy.utils.data.get_pkg_data_contents` (*data_name*, *encoding=None*, *cache=True*)
Retrieves a data file from the standard locations and returns its contents as a bytes object.

Parameters

data_name : str

Name/location of the desired data file. One of the following:

- The name of a data file included in the source distribution. The path is relative to the module calling this function. For example, if calling from `astropy.pkname`, use `'data/file.dat'` to get the file in `astropy/pkname/data/file.dat`. Double-dots can be used to go up a level. In the same example, use `'../data/file.dat'` to get `astropy/data/file.dat`.
- If a matching local file does not exist, the Astropy data server will be queried for the file.
- A hash like that produced by `compute_hash` can be requested, prefixed by `'hash/'` e.g. `'hash/395dd6493cc584df1e78b474fb150840'`. The hash will first be searched for locally, and if not found, the Astropy data server will be queried.
- A URL to some other file.

encoding : str, optional

When `None` (default), returns a file-like object with a `read` method that on Python 2.x returns `bytes` objects and on Python 3.x returns `str (unicode)` objects, using `locale.getpreferredencoding` as an encoding. This matches the default behavior of the built-in `open` when no `mode` argument is provided.

When `'binary'`, returns a file-like object where its `read` method returns `bytes` objects.

When another string, it is the name of an encoding, and the file-like object's `read` method will return `str (unicode)` objects, decoded from binary using the given encoding.

cache : bool

If True, the file will be downloaded and saved locally or the already-cached local copy will be accessed. If False, the file-like object will directly access the resource (e.g. if a remote URL is accessed, an object like that from `urllib2.urlopen` on Python 2 or `urllib.request.urlopen` on Python 3 is returned).

Returns

contents : bytes

The complete contents of the file as a bytes object.

Raises

urllib2.URLError, urllib.error.URLError

If a remote file cannot be found.

IOError

If problems occur writing or reading a local file.

See also:

`get_pkg_data_fileobj`

returns a file-like object with the data

`get_pkg_data_filename`

returns a local name for a file containing the data

`get_pkg_data_fileobjs`

`astropy.utils.data.get_pkg_data_fileobjs(datadir, pattern=u'*', encoding=None)`

Returns readable file objects for all of the data files in a given directory that match a given glob pattern.

Parameters

datadir : str

Name/location of the desired data files. One of the following:

- The name of a directory included in the source distribution. The path is relative to the module calling this function. For example, if calling from `astropy.pkname`, use `'data'` to get the files in `astropy/pkname/data`
- Remote URLs are not currently supported

pattern : str, optional

A UNIX-style filename glob pattern to match files. See the `glob` module in the standard library for more information. By default, matches all files.

encoding : str, optional

When `None` (default), returns a file-like object with a `read` method that on Python 2.x returns `bytes` objects and on Python 3.x returns `str (unicode)` objects, using `locale.getpreferredencoding` as an encoding. This matches the default behavior of the built-in `open` when no `mode` argument is provided.

When `'binary'`, returns a file-like object where its `read` method returns `bytes` objects.

When another string, it is the name of an encoding, and the file-like object's `read` method will return `str (unicode)` objects, decoded from binary using the given encoding.

Returns

fileobjs : iterator of file objects

File objects for each of the files on the local filesystem in *datadir* matching *pattern*.

Examples

This will retrieve the contents of the data file for the `astropy.wcs` tests:

```
from astropy.utils.data import get_pkg_data_filenames

for fd in get_pkg_data_filename('maps', '*.hdr'):
    fcontents = fd.read()
```

get_pkg_data_filenames

`astropy.utils.data.get_pkg_data_filenames` (*datadir*, *pattern=u'*'*)

Returns the path of all of the data files in a given directory that match a given glob pattern.

Parameters

datadir : str

Name/location of the desired data files. One of the following:

- The name of a directory included in the source distribution. The path is relative to the module calling this function. For example, if calling from `astropy.pkname`, use `'data'` to get the files in `astropy/pkname/data`.
- Remote URLs are not currently supported.

pattern : str, optional

A UNIX-style filename glob pattern to match files. See the `glob` module in the standard library for more information. By default, matches all files.

Returns

filenames : iterator of str

Paths on the local filesystem in *datadir* matching *pattern*.

Examples

This will retrieve the contents of the data file for the `astropy.wcs` tests:

```
from astropy.utils.data import get_pkg_data_filenames

for fn in get_pkg_data_filename('maps', '*.hdr'):
    with open(fn) as f:
        fcontents = f.read()
```

compute_hash

`astropy.utils.data.compute_hash` (*localfn*)

Computes the MD5 hash for a file.

The hash for a data file is used for looking up data files in a unique fashion. This is of particular use for tests; a test may require a particular version of a particular file, in which case it can be accessed via hash to get the appropriate version.

Typically, if you wish to write a test that requires a particular data file, you will want to submit that file to the astropy data servers, and use e.g. `get_pkg_data_filename('hash/a725fa6ba642587436612c2df0451956')`, but with the hash for your file in place of the hash in the example.

Parameters

localfn : str

The path to the file for which the hash should be generated.

Returns

md5hash : str

The hex digest of the MD5 hash for the contents of the `localfn` file.

`clear_download_cache`

`astropy.utils.data.clear_download_cache(hashorurl=None)`
Clears the data file cache by deleting the local file(s).

Parameters

hashorurl : str or None

If None, the whole cache is cleared. Otherwise, either specifies a hash for the cached file that is supposed to be deleted, or a URL that has previously been downloaded to the cache.

Raises

OSError

If the requested filename is not present in the data directory.

`get_free_space_in_dir`

`astropy.utils.data.get_free_space_in_dir(path)`
Given a path to a directory, returns the amount of free space (in bytes) on that filesystem.

Parameters

path : str

The path to a directory

Returns

bytes : int

The amount of free space on the partition that the directory is on.

`check_free_space_in_dir`

`astropy.utils.data.check_free_space_in_dir(path, size)`
Determines if a given directory has enough space to hold a file of a given size. Raises an IOError if the file would be too large.

Parameters

path : str

The path to a directory

size : int

A proposed filesize (in bytes)

Raises

IOError : There is not enough room on the filesystem

download_file

`astropy.utils.data.download_file(remote_url, cache=False, show_progress=True, timeout=None)`

Accepts a URL, downloads and optionally caches the result returning the filename, with a name determined by the file's MD5 hash. If `cache=True` and the file is present in the cache, just returns the filename.

Parameters

remote_url : str

The URL of the file to download

cache : bool, optional

Whether to use the cache

show_progress : bool, optional

Whether to display a progress bar during the download (default is `True`)

timeout : float, optional

The timeout, in seconds. Otherwise, use `astropy.utils.data.Conf.remote_timeout`.

Returns

local_path : str

Returns the local path that the file was download to.

Raises

urllib2.URLError, urllib.error.URLError

Whenever there's a problem getting the remote file.

download_files_in_parallel

`astropy.utils.data.download_files_in_parallel(urls, cache=False, show_progress=True, timeout=None)`

Downloads multiple files in parallel from the given URLs. Blocks until all files have downloaded. The result is a list of local file paths corresponding to the given urls.

Parameters

urls : list of str

The URLs to retrieve.

cache : bool, optional

Whether to use the cache

show_progress : bool, optional

Whether to display a progress bar during the download (default is `True`)

timeout : float, optional

Timeout for the requests in seconds (default is the configurable `astropy.utils.data.Conf.remote_timeout`).

Returns

paths : list of str

The local file paths corresponding to the downloaded URLs.

Classes

<code>Conf</code>	Configuration parameters for <code>astropy.utils.data</code> .
<code>CacheMissingWarning</code>	This warning indicates the standard cache directory is not accessible, with the first argument providing

Conf

class `astropy.utils.data.Conf`

Bases: `astropy.config.ConfigNamespace`

Configuration parameters for `astropy.utils.data`.

Attributes Summary

<code>compute_hash_block_size</code>	Block size for computing MD5 file hashes.
<code>dataurl</code>	URL for astropy remote data site.
<code>delete_temporary_downloads_at_exit</code>	If True, temporary download files created when the cache is inaccessible will be
<code>download_block_size</code>	Number of bytes of remote data to download per step.
<code>download_cache_lock_attempts</code>	Number of times to try to get the lock while accessing the data cache before givi
<code>remote_timeout</code>	Time to wait for remote data queries (in seconds).

Attributes Documentation

compute_hash_block_size

Block size for computing MD5 file hashes.

dataurl

URL for astropy remote data site.

delete_temporary_downloads_at_exit

If True, temporary download files created when the cache is inaccessible will be deleted at the end of the python session.

download_block_size

Number of bytes of remote data to download per step.

download_cache_lock_attempts

Number of times to try to get the lock while accessing the data cache before giving up.

remote_timeout

Time to wait for remote data queries (in seconds).

CacheMissingWarning

exception `astropy.utils.data.CacheMissingWarning`

This warning indicates the standard cache directory is not accessible, with the first argument providing the warning message. If `args[1]` is present, it is a filename indicating the path to a temporary file that was created to store a remote data download in the absence of the cache.

26.2.9 XML

The `astropy.utils.xml.*` modules provide various XML processing tools.

`astropy.utils.xml.check` Module

A collection of functions for checking various XML-related strings for standards compliance.

Functions

<code>check_anyuri(uri)</code>	Returns <code>True</code> if <code>uri</code> is a valid URI as defined in RFC 2396.
<code>check_id(ID)</code>	Returns <code>True</code> if <code>ID</code> is a valid XML ID.
<code>check_mime_content_type(content_type)</code>	Returns <code>True</code> if <code>content_type</code> is a valid MIME content type (syntactically at least).
<code>check_token(token)</code>	Returns <code>True</code> if <code>token</code> is a valid XML token, as defined by XML Schema Part 2.
<code>fix_id(ID)</code>	Given an arbitrary string, create one that can be used as an xml id.

`check_anyuri`

`astropy.utils.xml.check.check_anyuri(uri)`
Returns `True` if `uri` is a valid URI as defined in RFC 2396.

`check_id`

`astropy.utils.xml.check.check_id(ID)`
Returns `True` if `ID` is a valid XML ID.

`check_mime_content_type`

`astropy.utils.xml.check.check_mime_content_type(content_type)`
Returns `True` if `content_type` is a valid MIME content type (syntactically at least), as defined by RFC 2045.

`check_token`

`astropy.utils.xml.check.check_token(token)`
Returns `True` if `token` is a valid XML token, as defined by XML Schema Part 2.

`fix_id`

`astropy.utils.xml.check.fix_id(ID)`
Given an arbitrary string, create one that can be used as an xml id. This is rather simplistic at the moment, since it just replaces non-valid characters with underscores.

`astropy.utils.xml.iterparser` Module

This module includes a fast iterator-based XML parser.

Functions

<code>get_xml_iterator(*args, **kwargs)</code>	Returns an iterator over the elements of an XML file.
<code>get_xml_encoding(source)</code>	Determine the encoding of an XML file by reading its header.
<code>xml_readlines(source)</code>	Get the lines from a given XML file.

`get_xml_iterator`

`astropy.utils.xml.iterparser.get_xml_iterator(*args, **kwargs)`

Returns an iterator over the elements of an XML file.

The iterator doesn't ever build a tree, so it is much more memory and time efficient than the alternative in `cElementTree`.

Parameters

fd : readable file-like object or read function

Returns

parts : iterator

The iterator returns 4-tuples (*start, tag, data, pos*):

- *start*: when `True` is a start element event, otherwise an end element event.
- *tag*: The name of the element
- *data*: Depends on the value of *event*:
 - if *start* == `True`, data is a dictionary of attributes
 - if *start* == `False`, data is a string containing the text content of the element
- *pos*: Tuple (*line, col*) indicating the source of the event.

`get_xml_encoding`

`astropy.utils.xml.iterparser.get_xml_encoding(source)`

Determine the encoding of an XML file by reading its header.

Parameters

source : readable file-like object, read function or str path

Returns

encoding : str

`xml_readlines`

`astropy.utils.xml.iterparser.xml_readlines(source)`

Get the lines from a given XML file. Correctly determines the encoding and always returns unicode.

Parameters

source : readable file-like object, read function or str path

Returns

lines : list of unicode

`astropy.utils.xml.unescaper` Module

URL unescaper functions.

Functions

`unescape_all(url)` Recursively unescape a given URL.

unescape_all

`astropy.utils.xml.unescaper.unescape_all(url)`
Recursively unescape a given URL.

Note: ‘&&’ becomes a single ‘&’.

Parameters

url : str or bytes
URL to unescape.

Returns

clean_url : str or bytes
Unescaped URL.

astropy.utils.xml.validate Module

Functions to do XML schema and DTD validation. At the moment, this makes a subprocess call to xmllint. This could use a Python-based library at some point in the future, if something appropriate could be found.

Functions

`validate_schema(filename, schema_file)` Validates an XML file against a schema or DTD.

validate_schema

`astropy.utils.xml.validate.validate_schema(filename, schema_file)`
Validates an XML file against a schema or DTD.

Parameters

filename : str
The path to the XML file to validate

schema_file : str
The path to the XML schema or DTD

Returns

returncode, stdout, stderr : int, str, str
Returns the returncode from xmllint and the stdout and stderr as strings

astropy.utils.xml.writer Module

Contains a class that makes it simple to stream out well-formed and nicely-indented XML.

Classes

`XMLWriter(file)` A class to write well-formed and nicely indented XML.

XMLWriter

class `astropy.utils.xml.writer.XMLWriter(file)`
 A class to write well-formed and nicely indented XML.

Use like this:

```
w = XMLWriter(fh)
with w.tag('html'):
    with w.tag('body'):
        w.data('This is the content')
```

Which produces:

```
<html>
  <body>
    This is the content
  </body>
</html>
```

Parameters

file : writable file-like object.

Methods Summary

<code>close(id)</code>	Closes open elements, up to (and including) the element identified by the given identifier.
<code>comment(comment)</code>	Adds a comment to the output stream.
<code>data(text)</code>	Adds character data to the output stream.
<code>element(tag[, text, wrap, attrib])</code>	Adds an entire element.
<code>end([tag, indent, wrap])</code>	Closes the current element (opened by the most recent call to <code>start</code>).
<code>flush()</code>	
<code>get_indentation()</code>	Returns the number of indentation levels the file is currently in.
<code>get_indentation_spaces([offset])</code>	Returns a string of spaces that matches the current indentation level.
<code>object_attrs(obj, attrs)</code>	Converts an object with a bunch of attributes on an object into a dictionary for use by the <code>with</code> statement.
<code>start(tag[, attrib])</code>	Opens a new element.
<code>tag(*args, **kwds)</code>	A convenience method for use with the <code>with</code> statement:: <code>with writer.tag('foo'): writer.el</code>

Methods Documentation

close (*id*)

Closes open elements, up to (and including) the element identified by the given identifier.

Parameters

id : int

Element identifier, as returned by the `start` method.

comment (*comment*)

Adds a comment to the output stream.

Parameters

comment : str

Comment text, as a Unicode string.

data (*text*)

Adds character data to the output stream.

Parameters

text : str

Character data, as a Unicode string.

element (*tag, text=None, wrap=False, attrib={}, **extra*)

Adds an entire element. This is the same as calling `start`, `data`, and `end` in sequence. The `text` argument can be omitted.

end (*tag=None, indent=True, wrap=False*)

Closes the current element (opened by the most recent call to `start`).

Parameters

tag : str

Element name. If given, the tag must match the start tag. If omitted, the current element is closed.

flush ()

get_indentation ()

Returns the number of indentation levels the file is currently in.

get_indentation_spaces (*offset=0*)

Returns a string of spaces that matches the current indentation level.

static object_attrs (*obj, attrs*)

Converts an object with a bunch of attributes on an object into a dictionary for use by the `XMLWriter`.

Parameters

obj : object

Any Python object

attrs : sequence of str

Attribute names to pull from the object

Returns

attrs : dict

Maps attribute names to the values retrieved from `obj.attr`. If any of the attributes is `None`, it will not appear in the output dictionary.

start (*tag, attrib={}, **extra*)

Opens a new element. Attributes can be given as keyword arguments, or as a string/string dictionary. The method returns an opaque identifier that can be passed to the `close()` method, to close all open elements up to and including this one.

Parameters

tag : str

The element name

attrib : dict of str -> str

Attribute dictionary. Alternatively, attributes can be given as keyword arguments.

Returns

id : int

Returns an element identifier.

tag (*args, **kws)

A convenience method for use with the `with` statement:

```
with writer.tag('foo'):  
    writer.element('bar')  
# </foo> is implicitly closed here
```

Parameters are the same as to `start`.

Astropy project details

CURRENT STATUS OF SUB-PACKAGES

Astropy has benefited from the addition of widely tested legacy code, as well as new development, resulting in variations in stability across sub-packages. This document summarizes the current status of the Astropy sub-packages, so that users understand where they might expect changes in future, and which sub-packages they can safely use for production code.

Note that until version 1.0, even sub-packages considered *Mature* could undergo some user interface changes as we work to integrate the packages better. Thus, we cannot guarantee complete backward-compatibility between versions at this stage.

The classification is as follows:

The current planned and existing sub-packages are:

MAJOR RELEASE HISTORY

28.1 What’s New in Astropy 0.4?

28.1.1 Overview

Astropy 0.4 is a major release that adds new functionality since the 0.3.x series of releases. A new sub-package is included (see [SAMP](#)), a major overhaul of the [Coordinates](#) sub-package has been completed (see [Coordinates](#)), and many new features and improvements have been implemented for the existing sub-packages. In addition to usability improvements, we have made a number of changes in the infrastructure for setting up/installing the package (see [astropy-helpers](#) package), as well as reworking the configuration system (see [Configuration](#)).

In addition to these major changes, a large number of smaller improvements have occurred. Since v0.3, by the numbers:

- 819 issues have been closed
- 511 pull requests have been merged
- 57 distinct people have contributed code

28.1.2 Coordinates

The *Astronomical Coordinate Systems* ([astropy.coordinates](#)) sub-package has been largely re-designed based on broad community discussion and experience with v0.2 and v0.3. The key motivation was to implement coordinates within an extensible framework that cleanly separates the distinct aspects of data representation, coordinate frame representation and transformation, and user interface. This is described in the [APE5](#) document. Details of the new usage are given in the *Astronomical Coordinate Systems* ([astropy.coordinates](#)) section of the documentation.

An important point is that this sub-package is now considered stable and we do not expect any further major interface changes.

For most users the major change is that the recommended user interface to coordinate functionality is the [SkyCoord](#) class instead of classes like [ICRS](#) or [Galactic](#) (which are now called “frame” classes). For example:

```
>>> from astropy import units as u
>>> from astropy.coordinates import SkyCoord
>>> coordinate = SkyCoord(123.4*u.deg, 56.7*u.deg, frame='icrs')
```

The frame classes can still be used to create coordinate objects as before, but they are now more powerful because they can represent abstract coordinate frames without underlying data. The more typical use for frame classes is now:

```
>>> from astropy.coordinates import FK4 # Or ICRS, Galactic, or similar
>>> fk4_frame = FK4(equinox='J1980.0', obstime='2011-06-12T01:12:34')
```

```
>>> coordinate.transform_to(fk4_frame)
<SkyCoord (FK4): equinox=J1980.000, obstime=2011-06-12T01:12:34.000, ra=123.001698182 deg, dec=56.76>
```

At the lowest level of the framework are the representation classes which describe how to represent a point in a frame as a tuple of quantities, for instance as spherical, cylindrical, or cartesian coordinates. Any coordinate object can now be created using values in a number of common representations and be displayed using those representations. For example:

```
>>> coordinate = SkyCoord(1*u.pc, 2*u.pc, 3*u.pc, representation='cartesian')
>>> coordinate
<SkyCoord (ICRS): x=1.0 pc, y=2.0 pc, z=3.0 pc>

>>> coordinate.representation = 'physicsspherical'
>>> coordinate
<SkyCoord (ICRS): phi=63.4349488229 deg, theta=36.6992252005 deg, r=3.74165738677 pc>
```

28.1.3 SAMP

The *SAMP* (*Simple Application Messaging Protocol* (*astropy.vo.samp*) sub-package is a new sub-package (adapted from the *SAMPy* package) that contains an implementation of the Simple Application Messaging Protocol (SAMP) standard that allows communication with any SAMP-enabled application (such as TOPCAT, SAO Ds9, and Aladin). This sub-package includes both classes for a hub and a client, as well as an *integrated client* which automatically connects to any running SAMP hub and acts as a client:

```
>>> from astropy.vo.samp import SAMPIntegratedClient
>>> client = SAMPIntegratedClient()
>>> client.connect()
```

We can then use the client to communicate with other clients:

```
>>> client.get_registered_clients()
['hub', 'c1', 'c2']
>>> client.get_metadata('c1')
{'author.affiliation': 'Astrophysics Group, Bristol University',
 'author.email': 'm.b.taylor@bristol.ac.uk',
 'author.name': 'Mark Taylor',
 'home.page': 'http://www.starlink.ac.uk/topcat/',
 'samp.description.text': 'Tool for Operations on Catalogues And Tables',
 'samp.documentation.url': 'http://127.0.0.1:2525/doc/sun253/index.html',
 'samp.icon.url': 'http://127.0.0.1:2525/doc/images/tc_sok.gif',
 'samp.name': 'topcat',
 'topcat.version': '4.0-1'}
```

and we can then send for example tables and images over SAMP to other applications (see *SAMP (Simple Application Messaging Protocol)* (*astropy.vo.samp*) for examples of how to do this).

28.1.4 Quantity

The *Quantity* class has seen a series of optimizations and is now substantially faster. Additionally, the *time*, *coordinates*, and *table* subpackages integrate better with *Quantity*, with further improvements on the way for *table*. See *Quantity* and the other subpackage documentation sections for more details.

28.1.5 Inspecting FITS headers from the command line

The *FITS File handling* (`astropy.io.fits`) sub-package now provides a command line script for inspecting the header(s) of a FITS file. With Astropy 0.4 installed, run `fitsheader file.fits` in your terminal to print the header information to the screen in a human-readable format. Run `fitsheader --help` to see the full usage documentation.

28.1.6 Reading and writing HTML tables

The *ASCII Tables* (`astropy.io.ascii`) sub-package now provides the capability to read a table within an HTML file or web URL into an `astropy Table` object. This requires the `BeautifulSoup4` package to be installed. Conversely a `Table` object can now be written out as an HTML table.

28.1.7 Documentation URL changes

Starting in v0.4, the astropy documentation (and any package that uses `astropy-helpers`) will show the full name of functions and classes prefixed by the intended user-facing location. This is in contrast to previous versions, which pointed to the actual implementation module, rather than the intended public API location.

This will affect URLs pointing to specific documentation pages. For example, this URL points to the v0.3 location of the `astropy.cosmology.luminosity_distance` function:

- http://docs.astropy.org/en/v0.3/api/astropy.cosmology.funcs.luminosity_distance.html

while the appropriate URL for v0.4 and later is:

- http://docs.astropy.org/en/v0.4/api/astropy.cosmology.luminosity_distance.html

28.1.8 astropy-helpers package

We have now extracted our set-up and documentation utilities into a separate package, `astropy-helpers`. In practice, this does not change anything from a user point of view, but it is a big internal change that will allow any other packages to benefit from the set-up utilities developed for the core package without having to first install astropy.

28.1.9 Configuration

The configuration framework has been re-factored based on the design described in `APE3`. If you have previously edited the astropy configuration file (typically located at `~/.astropy/config/astropy.cfg`) then you should read over *Configuration transition* in order to understand how to update it to the new mechanism.

28.1.10 Deprecation and backward-incompatible changes

- Quantity comparisons with `==` or `!=` now always return `True` or `False`, even if units do not match (for which case a `UnitsError` used to be raised). [#2328]
- The functional interface for `astropy.cosmology` (e.g. `cosmology.H(z=0.5)`) is now deprecated in favor of the objected-oriented approach (`WMAP9.H(z=0.5)`). [#2343]
- The `astropy.coordinates` sub-package has undergone major changes for implementing the `APE5` plan for the package. A compatibility layer has been added that will allow common use cases of pre-v0.4 coordinates to work, but this layer will be removed in the next major version. Hence, any use of the coordinates package should be adapted to the new framework. Additionally, the compatibility layer cannot be used for convenience functions (like the `match_catalog_*()` functions), as these have been moved to `SkyCoord`. From this point on, major changes to the coordinates classes are not expected. [#2422]

- The configuration framework has been re-designed to the scheme of APE3. The previous framework based on `ConfigurationItem` is deprecated, and will be removed in a future release. Affiliated packages should update to the new configuration system, and any users who have customized their configuration file should migrate to the new configuration approach. Until they do, warnings will appear prompting them to do so.

28.1.11 Full change log

To see a detailed list of all changes in version 0.4 and prior, please see the *Full Changelog*.

28.1.12 Note on future versions

While the current release supports Python 2.6, 2.7, and 3.1 to 3.4, the next release (1.0) will drop support for Python 3.1 and 3.2.

28.2 What's New in Astropy 0.3?

See this page in the *Astropy v0.3 documentation*.

28.3 What's New in Astropy 0.2

See this page in the *Astropy v0.2 documentation*.

28.4 What's New in Astropy 0.1

This was the initial version of Astropy, released on June 19, 2012. It was released primarily as a “developer preview” for developers interested in working directly on Astropy, on affiliated packages, or on other software that might integrate with Astropy.

Astropy 0.1 integrated several existing packages under a single `astropy` package with a unified installer, including:

- `asciitable` as `astropy.io.ascii`
- `PyFITS` as `astropy.io.fits`
- `votable` as `astropy.io.vo`
- `PyWCS` as `astropy.wcs`

It also added the beginnings of the `astropy.cosmology` package, and new common data structures for science data in the `astropy.nddata` and `astropy.table` packages.

It also laid much of the groundwork for Astropy's installation and documentation frameworks, as well as tools for managing configuration and data management. These facilities are designed to be shared by Astropy's affiliated packages in the hopes of providing a framework on which other Astronomy-related Python packages can build.

KNOWN ISSUES

While most bugs and issues are managed using the [astropy issue tracker](#), this document lists issues that are too difficult to fix, may require some intervention from the user to workaround, or are due to bugs in other projects or packages.

29.1 Quantities lose their units with some operations

Quantities are subclassed from numpy's `ndarray` and in some numpy operations (and in scipy operations using numpy internally) the subclass is ignored, which means that either a plain array is returned, or a `Quantity` without units. E.g.:

```
In [1]: import astropy.units as u
```

```
In [2]: import numpy as np
```

```
In [3]: q = u.Quantity(np.arange(10.), u.m)
```

```
In [4]: np.dot(q, q)
```

```
Out[4]: 285.0
```

```
In [5]: np.hstack((q, q))
```

```
Out[5]:
```

```
<Quantity [ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9., 0., 1., 2., 3., 4.,  
          5., 6., 7., 8., 9.] (Unit not initialised)>
```

Work-arounds are available for some cases. For the above:

```
In [6]: q.dot(q)
```

```
Out[6]: <Quantity 285.0 m2>
```

```
In [7]: u.Quantity([q, q]).flatten()
```

```
Out[7]:
```

```
<Quantity [ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9., 0., 1., 2., 3., 4.,  
          5., 6., 7., 8., 9.] m>
```

See: <https://github.com/astropy/astropy/issues/1274>

29.2 Some docstrings can not be displayed in IPython < 0.13.2

Displaying long docstrings that contain Unicode characters may fail on some platforms in the IPython console (prior to IPython version 0.13.2):

```
In [1]: import astropy.units as u
```

```
In [2]: u.Angstrom?
```

```
ERROR: UnicodeEncodeError: 'ascii' codec can't encode character u'\xe5' in
position 184: ordinal not in range(128) [IPython.core.page]
```

This can be worked around by changing the default encoding to `utf-8` by adding the following to your `sitecustomize.py` file:

```
import sys
sys.setdefaultencoding('utf-8')
```

Note that in general, **this is not recommended**, because it can hide other Unicode encoding bugs in your application. However, in general if your application does not deal with text processing and you just want docstrings to work, this may be acceptable.

The IPython issue: <https://github.com/ipython/ipython/pull/2738>

29.3 Locale errors

On MacOS X, you may see the following error when running `setup.py`:

```
...
ValueError: unknown locale: UTF-8
```

You may also (on MacOS X or other platforms) see errors such as:

```
...
stderr = stderr.decode(stdio_encoding)
TypeError: decode() argument 1 must be str, not None
```

This is due to the `LC_CTYPE` environment variable being incorrectly set to `UTF-8` by default, which is not a valid locale setting. To fix this, set this environment variable, as well as the `LANG` and `LC_ALL` environment variables to e.g. `en_US.UTF-8` using, in the case of `bash`:

```
export LANG="en_US.UTF-8"
export LC_ALL="en_US.UTF-8"
export LC_CTYPE="en_US.UTF-8"
```

To avoid any issues in future, you should add this line to your e.g. `~/ .bash_profile` or `.bashrc` file.

To test these changes, open a new terminal and type `locale`, and you should see something like:

```
$ locale
LANG="en_US.UTF-8"
LC_COLLATE="en_US.UTF-8"
LC_CTYPE="en_US.UTF-8"
LC_MESSAGES="en_US.UTF-8"
LC_MONETARY="en_US.UTF-8"
LC_NUMERIC="en_US.UTF-8"
LC_TIME="en_US.UTF-8"
LC_ALL="en_US.UTF-8"
```

If so, you can go ahead and try running `setup.py` again (in the new terminal).

29.4 Floating point precision issues on Python 2.6 on Microsoft Windows

When converting floating point numbers to strings on Python 2.6 on a Microsoft Windows platform, some of the requested precision may be lost.

The easiest workaround is to install Python 2.7.

The Python issue: <http://bugs.python.org/issue7117>

29.5 Failing logging tests when running the tests in IPython

When running the Astropy tests using `astropy.test()` in an IPython interpreter some of the tests in the `astropy/tests/test_logger.py` fail. This is due to mutually incompatible behaviors in IPython and `py.test`, and is not due to a problem with the test itself or the feature being tested.

See: <https://github.com/astropy/astropy/issues/717>

29.6 mmap support for `astropy.io.fits` on GNU Hurd

On Hurd and possibly other platforms `flush()` on memory-mapped files is not implemented, so writing changes to a mmap'd FITS file may not be reliable and is thus disabled. Attempting to open a FITS file in writeable mode with `mmap` will result in a warning (and `mmap` will be disabled on the file automatically).

See: <https://github.com/astropy/astropy/issues/968>

29.7 Crash on upgrading from Astropy 0.2 to a newer version

It is possible for installation of a new version of Astropy, or upgrading of an existing installation to crash due to not having permissions on the `~/.astropy/` directory (in your home directory) or some file or subdirectory in that directory. In particular this can occur if you installed Astropy as the root user (such as with `sudo`) at any point. This can manifest in several ways, but the most common is a traceback ending with `ImportError: cannot import name config`. To resolve this issue either run `sudo chown -R <your_username> ~/.astropy` or, if you don't need anything in it you can blow it away with `sudo rm -rf ~/.astropy`.

See for example: <https://github.com/astropy/astropy/issues/987>

29.8 Color printing on Windows

Colored printing of log messages and other colored text does work in Windows but only when running in the IPython console. Colors are not currently supported in the basic Python command-line interpreter on Windows.

29.9 Table sorting can silently fail on MacOS X or Windows with Python 3 and Numpy < 1.6.2

In Python 3, prior to Numpy 1.6.2, there was a bug (in Numpy) that caused sorting of structured arrays to silently fail under certain circumstances (for example if the Table contains string columns) on MacOS X, Windows, and possibly

other platforms other than Linux. Since `Table.sort` relies on Numpy to internally sort the data, it is also affected by this bug. If you are using Python 3, and need the sorting functionality for tables, we recommend updating to a more recent version of Numpy.

29.10 Anaconda users should upgrade with conda, not pip

Upgrading Astropy in the anaconda python distribution using `pip` can result in a corrupted install with a mix of files from the old version and the new version. Anaconda users should update with `conda update astropy`. There may be a brief delay between the release of Astropy on PyPI and its release via the conda package manager; users can check the availability of new versions with `conda search astropy`.

29.11 Installation fails on Mageia-2 or Mageia-3 distributions

Building may fail with warning messages such as:

```
unable to find 'pow' or 'sincos'
```

at the linking phase. Upgrading the OS packages for Python should fix the issue, though an immediate workaround is to edit the file:

```
/usr/lib/python2.7/config/Makefile
```

and search for the line that adds the option `-Wl,--no-undefined` to the `LDFLAGS` variable and remove that option.

29.12 Remote data utilities in `astropy.utils.data` fail on some Python distributions

The remote data utilities in `astropy.utils.data` depend on the Python standard library `shelve` module, which in some cases depends on the standard library `bsddb` module. Some Python distributions, including but not limited to

- OS X, Python 2.7.5 via homebrew
- Linux, Python 2.7.6 via conda ¹
- Linux, Python 2.6.9 via conda

are built without support for the `bsddb` module, resulting in an error such as:

```
ImportError: No module named _bsddb
```

One workaround is to install the `bsddb3` module.

29.13 Very long integers in ASCII tables silently converted to float for Numpy 1.5

For Numpy 1.5, when reading an ASCII table that has integers which are too large to fit into the native C long int type for the machine, then the values get converted to float type with no warning. This is due to the behavior of

¹ Continuum says this will be fixed in their next Python build.

`numpy.array` and cannot easily be worked around. We recommend that users upgrade to a newer version of Numpy. For Numpy ≥ 1.6 a warning is printed and the values are treated as strings to preserve all information.

AUTHORS AND CREDITS

30.1 Astropy Project Coordinators

- Perry Greenfield
- Thomas Robitaille
- Erik Tollerud

30.2 Core Package Contributors

- Shailesh Ahuja
- Tom Aldcroft
- Kyle Barbary
- Geert Barentsen
- Paul Barrett
- Andreas Baumbach
- Chris Beaumont
- Daniel Bell
- Francesco Biscani
- Christopher Bonnett
- Médéric Boquien
- Larry Bradley
- Gustavo Bragança
- Erik M. Bray
- Eli Bressert
- Mabry Cervin
- Pritish Chakraborty
- Alex Conley
- Jean Connelly
- Simon Conseil

- Ryan Cooke
- Matt Craig
- Steven Crawford
- Neil Crighton
- Kelle Cruz
- Matt Davis
- Christoph Deil
- Nadia Dencheva
- Jörg Dietrich
- Axel Donath
- Michael Droettboom
- Zach Edwards
- Thomas Erben
- Henry Ferguson
- Jonathan Foster
- Ryan Fox
- Lehman Garrison
- Adam Ginsburg
- Christoph Gohlke
- Perry Greenfield
- Dylan Gregersen
- Frédéric Grollier
- Karan Grover
- Hans Moritz Günther
- Alex Hagen
- Emma Hogan
- Chris Hanley
- JC Hsu
- Marten van Kerkwijk
- Wolfgang Kerzendorf
- Lennard Kiehl
- Kacper Kowalik
- Roban Kramer
- Simon Liedtke
- Pey Lian Lim
- Serge Montagnac

- José Sabater Montes
- Michael Mueller
- Stuart Mumford
- Demitri Muna
- Prasanth Nair
- Bogdan Nicula
- Miruna Oprescu
- Luigi Paioro
- Asish Panda
- Madhura Parikh
- Sergio Pascual
- Rohit Patil
- David Perez-Suarez
- Ray Plante
- Adrian Price-Whelan
- Tanuj Rastogi
- Thomas Robitaille
- Juan Luis Cano Rodríguez
- Evert Rol
- Alex Rudy
- Joseph Ryan
- Eloy Salinas
- David Shiga
- David Shupe
- Leo Singer
- Brigitta Sipocz
- Shantanu Srivastava
- James Taylor
- Jeff Taylor
- Kirill Tchernyshyov
- Víctor Terrón
- Erik Tollerud
- James Turner
- Miguel de Val-Borro
- Jonathan Whitmore
- Benjamin Alan Weaver

- Julien Woillez
- Victor Zabalza

30.3 Other Credits

- Kyle Barbary for designing the Astropy logos and documentation themes.
- Andrew Pontzen and the `pynbody` team (For code that grew into `astropy.units`)
- Everyone on `astropy-dev` and the `astropy mailing list` for contributing to many discussions and decisions!

(If you have contributed to the Astropy project and your name is missing, please send an email to the coordinators, or open a pull request for this page in the astropy repository)

31.1 Astropy License

Astropy is licensed under a 3-clause BSD style license:

Copyright (c) 2011-2014, Astropy Developers

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the Astropy Team nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

31.2 Other Licenses

Full licenses for third-party software astropy is derived from or included with Astropy can be found in the `'licenses/'` directory of the source code distribution.

Part II

Getting help

If you want to get help or discuss issues with other Astropy users, you can sign up for the [astropy mailing list](#). Alternatively, the [astropy-dev mailing list](#) is where you should go to discuss more technical aspects of Astropy with the developers. You can also email the astropy developers privately at astropy-feedback@googlegroups.com...but remember that questions you ask publicly serve as resources for other users!

Part III

Reporting Issues

If you have found a bug in Astropy please report it. The preferred way is to create a new issue on the Astropy [GitHub issue page](#); that requires [creating a free account](#) on GitHub if you do not have one.

If you prefer not to create a GitHub account, please report the issue to either the [astropy mailing list](#), the [astropy-dev mailing list](#) or sending a private email to the astropy core developers at astropy-feedback@googlegroups.com.

Please include an example that demonstrates the issue that will allow the developers to reproduce and fix the problem. You may be asked to also provide information about your operating system and a full Python stack trace; the Astropy developers will walk you through obtaining a stack trace if it is necessary.

FOR ASTROPY-HELPERS

As of Astropy v0.4, Astropy and many affiliated packages use a package of utilities called astropy-helpers during building and installation. If you have any build/installation issue—particularly if you’re getting a traceback mentioning the `astropy_helpers` or `ah_bootstrap` modules—please send a report to the [astropy-helpers issue tracker](#). If you’re not sure, however, it’s fine to report via the main Astropy issue tracker or one of the other avenues described above.

Part IV

Contributing

The Astropy project is made both by and for its users, so we highly encourage contributions at all levels. This spans the gamut from sending an email mentioning a typo in the documentation or requesting a new feature all the way to developing a major new package.

The full range of ways to be part of the Astropy project are described at [Contribute to Astropy](#). To get started contributing code or documentation (no git or GitHub experience necessary):

TRY THE DEVELOPMENT VERSION

Note: `git` is the name of a source code management system. It is used to keep track of changes made to code and to manage contributions coming from several different people. If you want to read more about `git` right now take a look at [Git Basics](#).

If you have never used `git` before, allow one hour the first time you do this. You will not need to do this every time you want to contribute; most of it is one-time setup. You can count on one hand the number of `git` commands you will need to regularly use to keep your local copy of `Astropy` up to date. If you find this taking more than an hour email the [astropy developers list for help](#)

Trying out the development version of `Astropy` is useful in three ways:

- More users testing new features helps uncover bugs before the feature is released.
- A bug in the most recent stable release might have been fixed in the development version. Knowing whether that is the case can make your bug reports more useful.
- You will need to go through all of these steps before contributing any code to `Astropy`. Practicing now will save you time later if you plan to contribute.

33.1 Overview

Conceptually, there are several steps to getting a working copy of the latest version of `Astropy` on your computer:

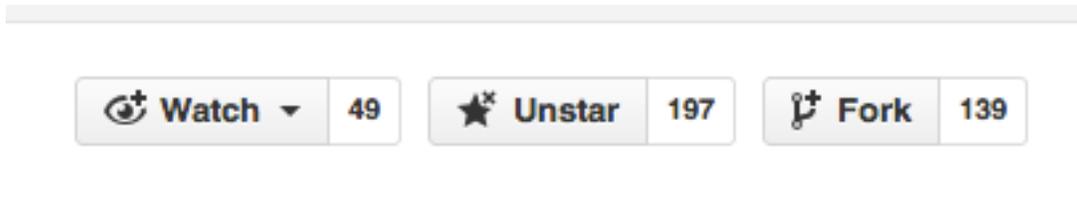
1. *Make your own copy of `Astropy` on GitHub*; this copy is called a *fork* (if you don't have an account on `github` yet, go there now and make one).
2. *Make sure `git` is installed and configured on your computer*
3. *Copy your fork of `Astropy` from GitHub to your computer*; this is called making a *clone* of the repository.
4. *Tell `git` where to look for changes in the development version of `Astropy`*
5. *Create your own private workspace*; this is called making a *branch*.
6. *“Activate” the development version of `astropy`*
7. *Test your development copy*
8. *Try out the development version*
9. *“Deactivate” the development version*

33.2 Step-by-step instructions

33.2.1 Make your own copy of Astropy on GitHub

In the language of [GitHub](#), making a copy of someone's code is called making a *fork*. A fork is a complete copy of the code and all of its revision history.

1. Log into your [GitHub](#) account.
2. Go to the [Astropy GitHub](#) home page.
3. Click on the *fork* button:



After a short pause and an animation of Octocat scanning a book on a flatbed scanner, you should find yourself at the home page for your own forked copy of [Astropy](#).

33.2.2 Make sure git is installed and configured on your computer

Check that git is installed:

Check by typing, in a terminal:

```
$ git --version
# if git is installed, will get something like: git version 1.8.4
```

If `git` is not installed, [get it](#).

Basic git configuration:

Follow the instructions at [Set Up Git at GitHub](#) to take care of two essential items:

- Set your user name and email in your copy of `git`
- Set up authentication so you don't have to type your github password every time you need to access github from the command line. The default method at [Set Up Git at GitHub](#) may require administrative privileges; if that is a problem, set up authentication [using SSH keys instead](#)

We also recommend setting up `git` so that when you copy changes from your computer to [GitHub](#) only the copy (called a *branch*) of `Astropy` that you are working on gets pushed up to [GitHub](#). *If* your version of `git` is 1.7.11 or, greater, you can do that with:

```
git config --global push.default simple
```

If you skip this step now it is not a problem; `git` will remind you to do it in those cases when it is relevant. If your version of `git` is less than 1.7.11, you can still continue without this, but it may lead to confusion later, as you might push up branches you do not intend to push.

Note: Make sure you make a note of which authentication method you set up because it affects the command you use to copy your [GitHub](#) fork to your computer.

If you set up password caching (the default method) the URLs will look like `https://github.com/your-user-name/astropy.git`.

If you set up SSH keys the URLs you use for making copies will look something like `git@github.com:your-user-name/astropy.git`.

33.2.3 Copy your fork of Astropy from GitHub to your computer

One of the commands below will make a complete copy of your GitHub fork of Astropy in a directory called `astropy`; which form you use depends on what kind of authentication you set up in the previous step:

```
# Use this form if you setup SSH keys...
$ git clone git@github.com:your-user-name/astropy.git
# ...otherwise use this form:
$ git clone https://github.com/your-user-name/astropy.git
```

If there is an error at this stage it is probably an error in setting up authentication.

33.2.4 Tell git where to look for changes in the development version of Astropy

Right now your local copy of Astropy doesn't know where the development version of Astropy is. There is no easy way to keep your local copy up to date. In `git` the name for another location of the same repository is a *remote*. The repository that contains the latest "official" development version is traditionally called the *upstream* remote, but here we use a more meaningful name for the remote: *astropy*.

Change into the `astropy` directory you created in the previous step and let `git` know about about the `astropy` remote:

```
cd astropy
git remote add astropy git://github.com/astropy/astropy.git
```

You can check that everything is set up properly so far by asking `git` to show you all of the remotes it knows about for your local repository of Astropy with `git remote -v`, which should display something like:

```
astropy  git://github.com/astropy/astropy.git (fetch)
astropy  git://github.com/astropy/astropy.git (push)
origin   git@github.com:your-user-name/astropy.git (fetch)
origin   git@github.com:your-user-name/astropy.git (push)
```

Note that `git` already knew about one remote, called *origin*; that is your fork of Astropy on GitHub.

To make more explicit that *origin* is really *your* fork of Astropy, rename that remote to your GitHub user name:

```
git remote rename origin your-user-name
```

33.2.5 Create your own private workspace

One of the nice things about `git` is that it is easy to make what is essentially your own private workspace to try out coding ideas. `git` calls these workspaces *branches*.

Your repository already has several branches; see them if you want by running `git branch -a`. Most of them are on `remotes/origin`; in other words, they exist on your remote copy of Astropy on GitHub.

There is one special branch, called *master*. Right now it is the one you are working on; you can tell because it has a marker next to it in your list of branches: `* master`.

To make a long story short, you never want to work on `master`. Always work on a branch.

To avoid potential confusion down the road, make your own branch now; this one you can call anything you like (when making contributions you should use a meaningful more name):

```
git branch my-own-astropy
```

You are *not quite* done yet. Git knows about this new branch; run `git branch` and you get:

```
* master
  my-own-astropy
```

The `*` indicates you are still working on master. To work on your branch instead you need to *check out* the branch `my-own-astropy`. Do that with:

```
git checkout my-own-astropy
```

and you should be rewarded with:

```
Switched to branch 'my-own-astropy'
```

33.2.6 “Activate” the development version of astropy

Right now you have the development version of [Astropy](#), but python will not see it. Though there are more sophisticated ways of managing multiple versions of [Astropy](#), for now this straightforward way will work (if you want to jump ahead to the more sophisticated method look at [virtual_envs](#)).

Note: There are a couple of circumstances in which this quick method of activating your copy of [Astropy](#) will NOT work and you need to go straight to using a virtual python environment:

- You use Python 3.
 - You want to work on C or Cython code in [Astropy](#).
-

In the directory where your copy of [Astropy](#) is type:

```
python setup.py develop
```

Several pages of output will follow the first time you do this; this wouldn't be a bad time to get a fresh cup of coffee. At the end of it you should see something like `Finished processing dependencies for astropy==0.3.dev6272`.

To make sure it has been activated **change to a different directory outside of the astropy distribution** and try this in python:

```
>>> import astropy
>>> astropy.__version__
'0.3.dev6272'
```

The actual version number will be different than in this example, but it should have dev in the name.

Warning: Right now every time you run Python, the development version of astropy will be used. That is fine for testing but you should make sure you change back to the stable version unless you are developing astropy. If you want to develop astropy, there is a better way of separating the development version from the version you do science with. That method, using a [virtualenv](#), is discussed at [virtual_envs](#). For now **remember to change back to your usual version** when you are done with this.

33.2.7 Test your development copy

Testing is an important part of making sure *Astropy* produces reliable, reproducible results. Before you try out a new feature or think you have found a bug make sure the tests run properly on your system.

If the test *don't* complete successfully, that is itself a bug—please [report it](#).

To run the tests, navigate back to the directory your copy of *astropy* is in on your computer, then, at the shell prompt, type:

```
python setup.py test
```

This is another good time to get some coffee or tea. The number of test is large. When the test are done running you will see a message something like this:

```
4741 passed, 85 skipped, 11 xfailed
```

Skips and xfails are fine, but if there are errors or failures please [report them](#).

33.2.8 Try out the development version

If you are going through this to ramp up to making more contributions to *Astropy* you don't actually have to do anything here.

If you are doing this because you have found a bug and are checking that it still exists in the development version, try running your code.

Or, just for fun, try out one of the [new features](#) in the development version.

Either way, once you are done, make sure you do the next step.

33.2.9 “Deactivate” the development version

Be sure to turn the development version off before you go back to doing science work with *Astropy*.

Navigate to the directory where your local copy of the development version is, then run:

```
python setup.py develop -u
```

You should really confirm it is deactivated by **changing to a different directory outside of the *astropy* distribution** and running this in python:

```
>>> import astropy
>>> astropy.__version__
'0.2.5'
```

The actual version number you see will likely be different than this example, but it should not have 'dev' in it.

HOW TO MAKE A CODE CONTRIBUTION

This document outlines the process for contributing code to the Astropy project.

Already experienced with git? Contributed before? Jump right to [Astropy Guidelines for git](#).

34.1 Pre-requisites

Before following the steps in this document you need:

- an account on [GitHub](#)
- a local copy of the astropy source. Instructions for doing that, including the basics you need for setting up git and GitHub, are at [Try the development version](#).

34.2 Strongly Recommended, but not required

You cannot easily work on the development version of astropy in a python environment in which you also use the stable version. It can be done — but can only be done *successfully* if you always remember whether the development version or stable version is the active one.

virtual_envs offer a better solution and take only a few minutes to set up. It is well worth your time.

Not sure what your first contribution should be? Take a look at the [Astropy issue list](#) and grab one labeled “easy”...but note that even your first “easy” fix is likely to take a while if you are not familiar with the Astropy source code! The developers are friendly and want you to help, so don’t be shy about asking questions on the [astropy-dev mailing list](#).

34.3 New to git?

34.3.1 Some git resources

If you have never used git or have limited experience with it, take a few minutes to look at these resources:

- [Interactive tutorial](#) that runs in a browser
- [Git Basics](#), part of a much longer [git book](#).

In practice, you need only a handful of [git](#) commands to make contributions to Astropy. There is a more extensive list of [git-resources](#) if you want more background.

34.3.2 Double check your setup

Before going further, make sure you have set up astropy as described in *Try the development version*.

In a terminal window, change directory to the one containing your clone of Astropy. Then, run `git remote`; the output should look something like this:

```
your-github-username
astropy
```

If that works, also run `git fetch --all`. If it runs without errors then your installation is working and you have a complete list of all branches in your clone, `your-github-username` and `astropy`.

34.3.3 About names in git

`git` is designed to be a *distributed* version control system. Each clone of a repository is, itself, a repository. That can lead to some confusion, especially for the branch called `master`. If you list all of the branches your clone of `git` knows about with `git branch -a` you will see there are *three* different branches called `master`:

```
* master                # this is master in your local repo
remotes/your-github-username/master # master on your fork of Astropy on GitHub
remotes/astropy/master   # the official development branch of Astropy
```

The naming scheme used by `git` will also be used here. A plain branch name, like `master` means a branch in your local copy of Astropy. A branch on a remote, like `astropy`, is labeled by that remote, `astropy/master`.

This duplication of names can get very confusing for maintainers when trying to merge code contributions into the official `master` branch, `astropy/master`. As a result, you should never do any work in your `master` branch, `master`. Always work on a branch instead.

34.3.4 Essential git commands

A full `git` tutorial is beyond the scope of this document but this list describes the few `git` commands you are likely to encounter in contributing to Astropy:

- `git fetch` gets the latest development version of Astropy, which you will use as the basis for making your changes.
- `git branch` makes a logically separate copy of Astropy to keep track of your changes.
- `git add` stages files you have changed or created for addition to `git`.
- `git commit` adds your staged changes to the repository.
- `git push` copies the changes you committed to GitHub
- `git status` to see a list of files that have been modified or created.

Note: A good graphical interface to `git` makes some of these steps much easier. Some options are described in *git_gui_options*.

34.3.5 If something goes wrong

`git` provides a number of ways to recover from errors. If you end up making a `git` mistake, do not hesitate to ask for help. An additional resource that walks you through recovering from `git` mistakes is the *git choose-your-own-adventure*.

34.4 Astropy Guidelines for git

- Don't use your `master` branch for anything.
- Make a new branch, called a *feature branch*, for each separable set of changes: “one task, one branch” (ipython git workflow).
- Start that new *feature branch* from the most current development version of astropy (instructions are below).
- Name your branch for the purpose of the changes, for example `bugfix-for-issue-14` or `refactor-database-code`.
- Make frequent commits, and always include a commit message. Each commit should represent one logical set of changes.
- Ask on the [astropy-dev mailing list](#) if you get stuck.
- Never merge changes from `astropy/master` into your feature branch. If changes in the development version require changes to our code you can *Rebase, but only if asked*.

In addition there are a couple of `git` naming conventions used in this document:

- Change the name of the remote `origin` to `your-github-username`.
- Name the remote that is the primary Astropy repository `astropy`; in prior versions of this documentation it was referred to as `upstream`.

34.5 Workflow

These, conceptually, are the steps you will follow in contributing to Astropy:

1. *Fetch the latest Astropy*
2. *Make a new feature branch*; you will make your changes on this branch.
3. *Install your branch*
4. Follow *The editing workflow* to write/edit/document/test code - make frequent, small commits.
5. *Add a changelog entry*
6. *Copy your changes to GitHub*
7. From GitHub, *Ask for your changes to be reviewed* to let the Astropy maintainers know you have contributions to review.
8. *Revise and push as necessary* in response to comments on the pull request. Pushing those changes to GitHub automatically updates the pull request.

This way of working helps to keep work well organized, with readable history. This in turn makes it easier for project maintainers (that might be you) to see what you've done, and why you did it.

A worked example that follows these steps for fixing an Astropy issue is at [astropy-fix-example](#).

Some additional topics related to `git` are in *additional-git*.

34.6 Fetch the latest Astropy

From time to time you should fetch the development version (i.e. Astropy `astropy/master`) changes from GitHub:

```
git fetch astropy
```

This will pull down any commits you don't have, and set the remote branches to point to the latest commit. For example, 'trunk' is the branch referred to by `astropy/master`, and if there have been commits since you last checked, `astropy/master` will change after you do the fetch.

34.7 Make a new feature branch

34.7.1 Make the new branch

When you are ready to make some changes to the code, you should start a new branch. Branches that are for a collection of related edits are often called 'feature branches'.

Making a new branch for each set of related changes will make it easier for someone reviewing your branch to see what you are doing.

Choose an informative name for the branch to remind yourself and the rest of us what the changes in the branch are for. Branch names like `add-ability-to-fly` or `buxfix-for-issue-42` clearly describe the purpose of the branch.

Always make your branch from `astropy/master` so that you are basing your changes on the latest version of Astropy:

```
# Update the mirror of trunk
git fetch astropy

# Make new feature branch starting at astropy/master
git branch my-new-feature astropy/master
git checkout my-new-feature
```

34.7.2 Connect the branch to GitHub

At this point you have made and checked out a new branch, but `git` does not know it should be connected to your fork on GitHub. You need that connection for your proposed changes to be managed by the Astropy maintainers on GitHub.

To connect your local branch to GitHub, you `git push` this new branch up to your GitHub repo with the `--set-upstream` option:

```
git push --set-upstream your-github-username my-new-feature
```

From now on `git` will know that `my-new-feature` is related to the `your-github-username/my-new-feature` branch in your GitHub fork of Astropy.

You will still need to `git push` your changes to GitHub periodically. The setup in this section will make that easier.

34.8 Install your branch

Ideally you should set up a python virtual environment just for this fix; instructions for doing to are at *virtual_envs*. Doing so ensures you will not corrupt your main astropy install and makes it very easy to recover from mistakes.

Once you have activated that environment you need to install the version of Astropy you are working on. Do that with:

```
python setup.py develop # typically python 2.x, not python 3
```

or:

```
python3 setup.py install # python 3...
# ...though python3 may be called python3.3 or just python,
# depending on your system.
```

If you are using python 3 you will need to re-install after making changes to the Astropy source code. Re-installing goes much faster than the initial install because it typically does not require new compilation.

34.9 The editing workflow

Conceptually, you will:

1. Make changes to one or more files and/or add a new file.
2. Check that your changes do not break existing code.
3. Add documentation to your code and, as appropriate, to the Astropy documentation.
4. Ideally, also make sure your changes do not break the documentation.
5. Add tests of the code you contribute.
6. Commit your changes in `git`
7. Repeat as necessary.

34.9.1 In more detail

1. Make some changes to one or more files. You should follow the Astropy *Coding Guidelines*. Each logical set of changes should be treated as one commit. For example, if you are fixing a known bug in Astropy and notice a different bug while implementing your fix, implement the fix to that new bug as a different set of changes.
2. Test that your changes do not lead to *regressions*, i.e. that your changes do not break existing code, by running the Astropy tests. You can run all of the Astropy tests from `ipython` with:

```
import astropy
astropy.test()
```

If your change involves only a small part of Astropy, e.g. `Time`, you can run just those tests:

```
import astropy
astropy.test('time')
```

3. Make sure your code includes appropriate docstrings, described at *Astropy Docstring Rules*. If appropriate, as when you are adding a new feature, you should update the appropriate documentation in the `docs` directory; a detailed description is in *Writing Documentation*.
4. If you have `sphinx` installed, you can also check that the documentation builds and looks correct by running, from the `astropy` directory:

```
python setup.py build_sphinx
```

The last line should just state `build succeeded`, and should not mention any warnings. (For more details, see *Writing Documentation*.)

5. Add tests of your new code, if appropriate. Some changes (e.g. to documentation) do not need tests. Detailed instructions are at *Testing Guidelines*, but if you have no experience writing tests or with the `py.test` testing framework submit your changes without adding tests, but mention in the pull request that you have not written tests. An example of writing a test is in *astropy-fix-example*.
6. Stage your changes using `git add` and commit them using `git commit`. An example of doing that, based on the fix for an actual Astropy issue, is at *astropy-fix-example*.

Note: Make your `git` commit messages short and descriptive. If a commit fixes an issue, include, on the second or later line of the commit message, the issue number in the commit message, like this: `Closes #123`. Doing so will automatically close the issue when the pull request is accepted.

7. Some modifications require more than one commit; if in doubt, break your changes into a few, smaller, commits rather than one large commit that does many things at once. Repeat the steps above as necessary!

34.10 Add a changelog entry

Add an entry to the file `CHANGES.rst` briefly describing the change you made. Include the pull request number if the change fixes an issue. An example entry, for the changes which fixed [issue 1845](#), is:

```
- `astropy.wcs.Wcs.printwcs` will no longer warn that `cdelt` is
  being ignored when none was present in the FITS file. [#1845]
```

If the change is a new feature, rather than an existing issue, you will not be able to put in the issue number until *after* you make the pull request.

34.11 Copy your changes to GitHub

This step is easy because of the way you created the feature branch. Just:

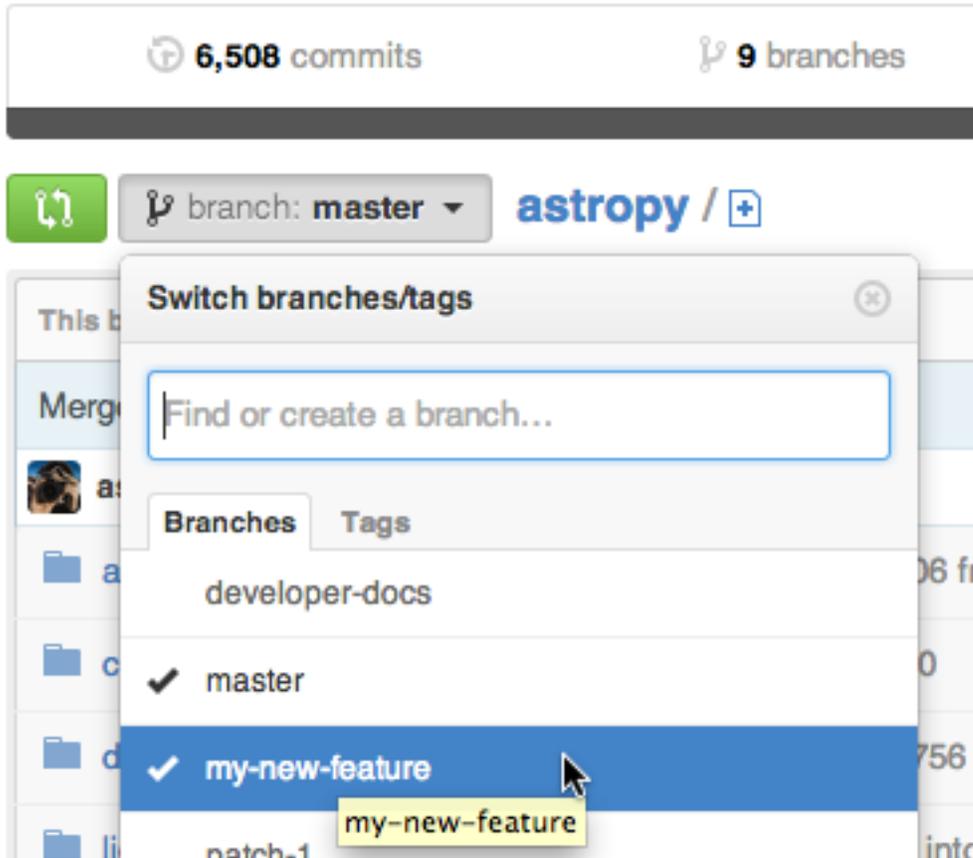
```
git push
```

34.12 Ask for your changes to be reviewed

A *pull request* on GitHub is a request to merge the changes you have made into another repository.

When you are ready to ask for someone to review your code and consider merging it into Astropy:

1. Go to the URL of your fork of Astropy, e.g., <https://github.com/your-user-name/astropy>.
2. Use the 'Switch Branches' dropdown menu to select the branch with your changes:



3. Click on the ‘Pull request’ button:



Enter a title for the set of changes, and some explanation of what you’ve done. If there is anything you’d like particular attention for, like a complicated change or some code you are not happy with, add the details here.

If you don’t think your request is ready to be merged, just say so in your pull request message. This is still a good way to start a preliminary code review.

34.13 Revise and push as necessary

You may be asked to make changes in the discussion of the pull request. Make those changes in your local copy, commit them to your local repo and push them to GitHub. GitHub will automatically update your pull request.

34.14 Rebase, but only if asked

Sometimes the maintainers of Astropy will ask you to *rebase* your changes before they are merged into the main Astropy repository.

Conceptually, rebasing means taking your changes and applying them to the latest version of the development branch of the official astropy as though that was the version you had originally branched from.

Behind the scenes, `git` is deleting the changes and branch you made, making the changes others made to the development branch of Astropy, then re-making your branch from the development branch and applying your changes to your branch. This results in re-writing the history of commits, which is why you should do it only if asked.

It is easier to make mistakes rebasing than other areas of `git`, so before you start make a branch to serve as a backup copy of your work:

```
git branch tmp my-new-feature # make temporary branch--will be deleted later
```

The actual rebasing is usually easy:

```
git fetch astropy/master # get the latest development astropy
git rebase astropy/master my-new-feature
```

You are more likely to run into *conflicts* here—places where the changes you made conflict with changes that someone else made—than anywhere else. Ask for help if you need it.

After the rebase you need to push your changes to GitHub; you will need force the push because `git` objects to re-writing the history of the repository after you have pushed it somewhere:

```
git push -f
```

If you run into any problems, do not hesitate to ask. A more detailed conceptual discussing of rebasing is at *rebase-on-trunk*.

Once your rebase is successfully pushed to GitHub you can delete the backup branch you made:

```
git branch -D tmp
```

Part V

Developer Documentation

The developer documentation contains instructions for how to contribute to Astropy or affiliated packages, as well as coding, documentation, and testing guidelines. For the guiding vision of this process and the project as a whole, see [development/vision](#).

HOW TO MAKE A CODE CONTRIBUTION

This document outlines the process for contributing code to the Astropy project.

Already experienced with git? Contributed before? Jump right to *Astropy Guidelines for git*.

35.1 Pre-requisites

Before following the steps in this document you need:

- an account on [GitHub](#)
- a local copy of the astropy source. Instructions for doing that, including the basics you need for setting up git and GitHub, are at *Try the development version*.

35.2 Strongly Recommended, but not required

You cannot easily work on the development version of astropy in a python environment in which you also use the stable version. It can be done — but can only be done *successfully* if you always remember whether the development version or stable version is the active one.

virtual_envs offer a better solution and take only a few minutes to set up. It is well worth your time.

Not sure what your first contribution should be? Take a look at the [Astropy issue list](#) and grab one labeled “easy”...but note that even your first “easy” fix is likely to take a while if you are not familiar with the Astropy source code! The developers are friendly and want you to help, so don’t be shy about asking questions on the [astropy-dev mailing list](#).

35.3 New to git?

35.3.1 Some git resources

If you have never used git or have limited experience with it, take a few minutes to look at these resources:

- [Interactive tutorial](#) that runs in a browser
- [Git Basics](#), part of a much longer [git book](#).

In practice, you need only a handful of [git](#) commands to make contributions to Astropy. There is a more extensive list of *git-resources* if you want more background.

35.3.2 Double check your setup

Before going further, make sure you have set up astropy as described in *Try the development version*.

In a terminal window, change directory to the one containing your clone of Astropy. Then, run `git remote`; the output should look something like this:

```
your-github-username
astropy
```

If that works, also run `git fetch --all`. If it runs without errors then your installation is working and you have a complete list of all branches in your clone, `your-github-username` and `astropy`.

35.3.3 About names in git

`git` is designed to be a *distributed* version control system. Each clone of a repository is, itself, a repository. That can lead to some confusion, especially for the branch called `master`. If you list all of the branches your clone of `git` knows about with `git branch -a` you will see there are *three* different branches called `master`:

```
* master                # this is master in your local repo
remotes/your-github-username/master # master on your fork of Astropy on GitHub
remotes/astropy/master   # the official development branch of Astropy
```

The naming scheme used by `git` will also be used here. A plain branch name, like `master` means a branch in your local copy of Astropy. A branch on a remote, like `astropy`, is labeled by that remote, `astropy/master`.

This duplication of names can get very confusing for maintainers when trying to merge code contributions into the official `master` branch, `astropy/master`. As a result, you should never do any work in your `master` branch, `master`. Always work on a branch instead.

35.3.4 Essential git commands

A full `git` tutorial is beyond the scope of this document but this list describes the few `git` commands you are likely to encounter in contributing to Astropy:

- `git fetch` gets the latest development version of Astropy, which you will use as the basis for making your changes.
- `git branch` makes a logically separate copy of Astropy to keep track of your changes.
- `git add` stages files you have changed or created for addition to `git`.
- `git commit` adds your staged changes to the repository.
- `git push` copies the changes you committed to GitHub
- `git status` to see a list of files that have been modified or created.

Note: A good graphical interface to `git` makes some of these steps much easier. Some options are described in *git_gui_options*.

35.3.5 If something goes wrong

`git` provides a number of ways to recover from errors. If you end up making a `git` mistake, do not hesitate to ask for help. An additional resource that walks you through recovering from `git` mistakes is the `git choose-your-own-adventure`.

35.4 Astropy Guidelines for git

- Don't use your `master` branch for anything.
- Make a new branch, called a *feature branch*, for each separable set of changes: “one task, one branch” (ipython git workflow).
- Start that new *feature branch* from the most current development version of astropy (instructions are below).
- Name your branch for the purpose of the changes, for example `bugfix-for-issue-14` or `refactor-database-code`.
- Make frequent commits, and always include a commit message. Each commit should represent one logical set of changes.
- Ask on the [astropy-dev mailing list](#) if you get stuck.
- Never merge changes from `astropy/master` into your feature branch. If changes in the development version require changes to our code you can *Rebase, but only if asked*.

In addition there are a couple of `git` naming conventions used in this document:

- Change the name of the remote `origin` to `your-github-username`.
- Name the remote that is the primary Astropy repository `astropy`; in prior versions of this documentation it was referred to as `upstream`.

35.5 Workflow

These, conceptually, are the steps you will follow in contributing to Astropy:

1. *Fetch the latest Astropy*
2. *Make a new feature branch*; you will make your changes on this branch.
3. *Install your branch*
4. Follow *The editing workflow* to write/edit/document/test code - make frequent, small commits.
5. *Add a changelog entry*
6. *Copy your changes to GitHub*
7. From GitHub, *Ask for your changes to be reviewed* to let the Astropy maintainers know you have contributions to review.
8. *Revise and push as necessary* in response to comments on the pull request. Pushing those changes to GitHub automatically updates the pull request.

This way of working helps to keep work well organized, with readable history. This in turn makes it easier for project maintainers (that might be you) to see what you've done, and why you did it.

A worked example that follows these steps for fixing an Astropy issue is at [astropy-fix-example](#).

Some additional topics related to `git` are in *additional-git*.

35.6 Fetch the latest Astropy

From time to time you should fetch the development version (i.e. Astropy `astropy/master`) changes from GitHub:

```
git fetch astropy
```

This will pull down any commits you don't have, and set the remote branches to point to the latest commit. For example, 'trunk' is the branch referred to by `astropy/master`, and if there have been commits since you last checked, `astropy/master` will change after you do the fetch.

35.7 Make a new feature branch

35.7.1 Make the new branch

When you are ready to make some changes to the code, you should start a new branch. Branches that are for a collection of related edits are often called 'feature branches'.

Making a new branch for each set of related changes will make it easier for someone reviewing your branch to see what you are doing.

Choose an informative name for the branch to remind yourself and the rest of us what the changes in the branch are for. Branch names like `add-ability-to-fly` or `buxfix-for-issue-42` clearly describe the purpose of the branch.

Always make your branch from `astropy/master` so that you are basing your changes on the latest version of Astropy:

```
# Update the mirror of trunk
git fetch astropy

# Make new feature branch starting at astropy/master
git branch my-new-feature astropy/master
git checkout my-new-feature
```

35.7.2 Connect the branch to GitHub

At this point you have made and checked out a new branch, but `git` does not know it should be connected to your fork on GitHub. You need that connection for your proposed changes to be managed by the Astropy maintainers on GitHub.

To connect your local branch to GitHub, you `git push` this new branch up to your GitHub repo with the `--set-upstream` option:

```
git push --set-upstream your-github-username my-new-feature
```

From now on `git` will know that `my-new-feature` is related to the `your-github-username/my-new-feature` branch in your GitHub fork of Astropy.

You will still need to `git push` your changes to GitHub periodically. The setup in this section will make that easier.

35.8 Install your branch

Ideally you should set up a python virtual environment just for this fix; instructions for doing to are at *virtual_envs*. Doing so ensures you will not corrupt your main astropy install and makes it very easy to recover from mistakes.

Once you have activated that environment you need to install the version of Astropy you are working on. Do that with:

```
python setup.py develop # typically python 2.x, not python 3
```

or:

```
python3 setup.py install # python 3...
# ...though python3 may be called python3.3 or just python,
# depending on your system.
```

If you are using python 3 you will need to re-install after making changes to the Astropy source code. Re-installing goes much faster than the initial install because it typically does not require new compilation.

35.9 The editing workflow

Conceptually, you will:

1. Make changes to one or more files and/or add a new file.
2. Check that your changes do not break existing code.
3. Add documentation to your code and, as appropriate, to the Astropy documentation.
4. Ideally, also make sure your changes do not break the documentation.
5. Add tests of the code you contribute.
6. Commit your changes in `git`
7. Repeat as necessary.

35.9.1 In more detail

1. Make some changes to one or more files. You should follow the Astropy *Coding Guidelines*. Each logical set of changes should be treated as one commit. For example, if you are fixing a known bug in Astropy and notice a different bug while implementing your fix, implement the fix to that new bug as a different set of changes.
2. Test that your changes do not lead to *regressions*, i.e. that your changes do not break existing code, by running the Astropy tests. You can run all of the Astropy tests from `ipython` with:

```
import astropy
astropy.test()
```

If your change involves only a small part of Astropy, e.g. `Time`, you can run just those tests:

```
import astropy
astropy.test('time')
```

3. Make sure your code includes appropriate docstrings, described at *Astropy Docstring Rules*. If appropriate, as when you are adding a new feature, you should update the appropriate documentation in the `docs` directory; a detailed description is in *Writing Documentation*.
4. If you have `sphinx` installed, you can also check that the documentation builds and looks correct by running, from the `astropy` directory:

```
python setup.py build_sphinx
```

The last line should just state `build succeeded`, and should not mention any warnings. (For more details, see *Writing Documentation*.)

5. Add tests of your new code, if appropriate. Some changes (e.g. to documentation) do not need tests. Detailed instructions are at *Testing Guidelines*, but if you have no experience writing tests or with the `py.test` testing framework submit your changes without adding tests, but mention in the pull request that you have not written tests. An example of writing a test is in *astropy-fix-example*.
6. Stage your changes using `git add` and commit them using `git commit`. An example of doing that, based on the fix for an actual Astropy issue, is at *astropy-fix-example*.

Note: Make your `git` commit messages short and descriptive. If a commit fixes an issue, include, on the second or later line of the commit message, the issue number in the commit message, like this: `Closes #123`. Doing so will automatically close the issue when the pull request is accepted.

7. Some modifications require more than one commit; if in doubt, break your changes into a few, smaller, commits rather than one large commit that does many things at once. Repeat the steps above as necessary!

35.10 Add a changelog entry

Add an entry to the file `CHANGES.rst` briefly describing the change you made. Include the pull request number if the change fixes an issue. An example entry, for the changes which fixed [issue 1845](#), is:

```
- `astropy.wcs.Wcs.printwcs` will no longer warn that `cdelt` is
  being ignored when none was present in the FITS file. [#1845]
```

If the change is a new feature, rather than an existing issue, you will not be able to put in the issue number until *after* you make the pull request.

35.11 Copy your changes to GitHub

This step is easy because of the way you created the feature branch. Just:

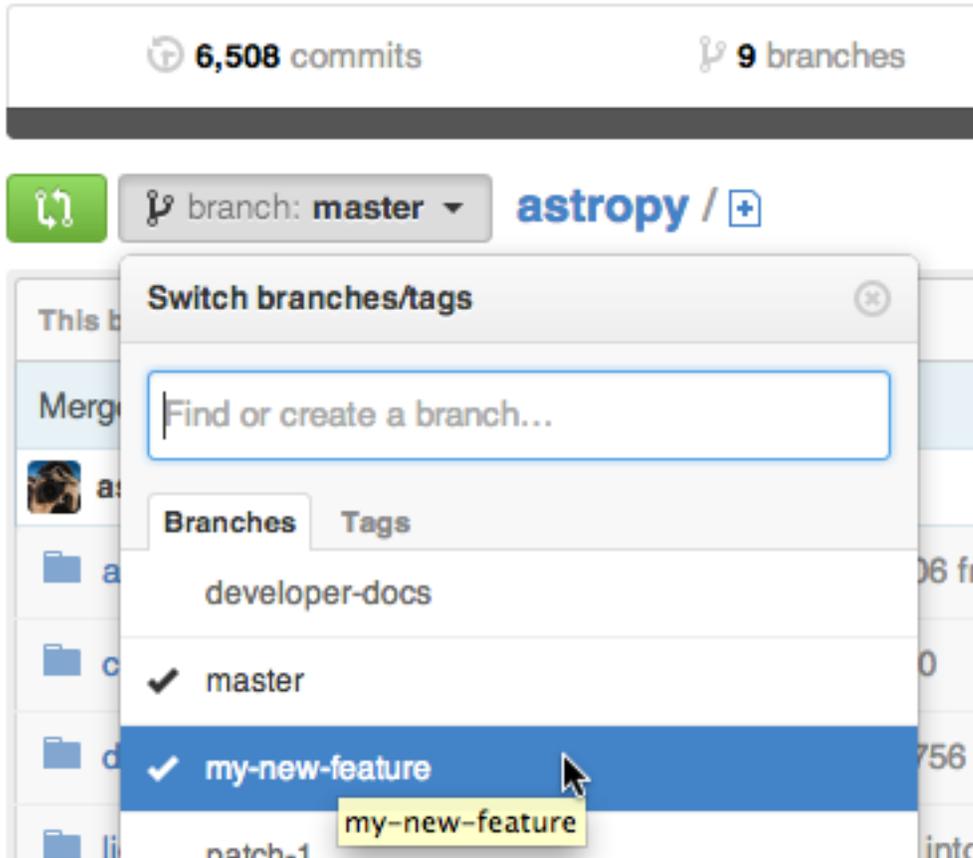
```
git push
```

35.12 Ask for your changes to be reviewed

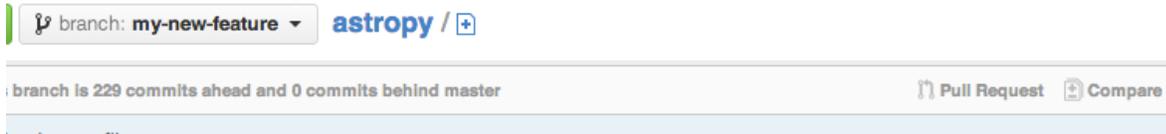
A *pull request* on GitHub is a request to merge the changes you have made into another repository.

When you are ready to ask for someone to review your code and consider merging it into Astropy:

1. Go to the URL of your fork of Astropy, e.g., <https://github.com/your-user-name/astropy>.
2. Use the 'Switch Branches' dropdown menu to select the branch with your changes:



3. Click on the ‘Pull request’ button:



Enter a title for the set of changes, and some explanation of what you’ve done. If there is anything you’d like particular attention for, like a complicated change or some code you are not happy with, add the details here.

If you don’t think your request is ready to be merged, just say so in your pull request message. This is still a good way to start a preliminary code review.

35.13 Revise and push as necessary

You may be asked to make changes in the discussion of the pull request. Make those changes in your local copy, commit them to your local repo and push them to GitHub. GitHub will automatically update your pull request.

35.14 Rebase, but only if asked

Sometimes the maintainers of Astropy will ask you to *rebase* your changes before they are merged into the main Astropy repository.

Conceptually, rebasing means taking your changes and applying them to the latest version of the development branch of the official astropy as though that was the version you had originally branched from.

Behind the scenes, `git` is deleting the changes and branch you made, making the changes others made to the development branch of Astropy, then re-making your branch from the development branch and applying your changes to your branch. This results in re-writing the history of commits, which is why you should do it only if asked.

It is easier to make mistakes rebasing than other areas of `git`, so before you start make a branch to serve as a backup copy of your work:

```
git branch tmp my-new-feature # make temporary branch--will be deleted later
```

The actual rebasing is usually easy:

```
git fetch astropy/master # get the latest development astropy
git rebase astropy/master my-new-feature
```

You are more likely to run into *conflicts* here—places where the changes you made conflict with changes that someone else made—than anywhere else. Ask for help if you need it.

After the rebase you need to push your changes to GitHub; you will need force the push because `git` objects to re-writing the history of the repository after you have pushed it somewhere:

```
git push -f
```

If you run into any problems, do not hesitate to ask. A more detailed conceptual discussing of rebasing is at *rebase-on-trunk*.

Once your rebase is successfully pushed to GitHub you can delete the backup branch you made:

```
git branch -D tmp
```

CODING GUIDELINES

This section describes requirements and guidelines that should be followed both for the core package and for affiliated packages.

Note: Affiliated packages will only be considered for integration as a module in the core package once these guidelines have been followed.

36.1 Interface and Dependencies

- All code must be compatible with Python 2.6, 2.7, as well as 3.1 and later. The use of `six` for writing code that is portable between Python 2.x and 3.x is encouraged going forward. However, much of our legacy code still uses `2to3` to process Python 2.x files to be compatible with Python 3.x.

Packages that use `six` must include the following in their `setup_package.py` file:

```
def requires_2to3():  
    return False
```

Code that uses `six` should use the following preamble:

```
from __future__ import (absolute_import, division, print_function,  
                        unicode_literals)
```

Code that uses `2to3` may limit the `__future__` statement to only the following, if necessary (though using all 4 is still preferred):

```
from __future__ import print_function, division
```

Additional information on writing code using `six` that is compatible with both Python 2.x and 3.x is in the section *Writing portable code for Python 2 and 3*.

- The new Python 3 formatting style should be used (i.e. `{0:s}.format("spam")` instead of `%s" % "spam"`), although when using positional arguments, the position should always be specified (i.e. `{:s}` is not compatible with Python 2.6).
- The core package and affiliated packages should be importable with no dependencies other than components already in the Astropy core, the [Python Standard Library](#), and [NumPy 1.5.1](#) or later.
- The package should be importable from the source tree at build time. This means that, for example, if the package relies on C extensions that have yet to be built, the Python code is still importable, even if none of its functionality will work. One way to ensure this is to import the functions in the C extensions only within the functions/methods that require them (see next bullet point).

- Additional dependencies - such as `SciPy`, `Matplotlib`, or other third-party packages - are allowed for sub-modules or in function calls, but they must be noted in the package documentation and should only affect the relevant component. In functions and methods, the optional dependency should use a normal `import` statement, which will raise an `ImportError` if the dependency is not available.

At the module level, one can subclass a class from an optional dependency like so:

```
try:
    from opdep import Superclass
except ImportError:
    warn(AstropyWarning('opdep is not present, so <functionality below> will not work.'))
    class SuperClass(object): pass

class Whatever(Superclass):
    ...
```

- General utilities necessary for but not specific to the package or sub-package should be placed in the `packagename.utils` module. These utilities will be moved to the `astropy.utils` module when the package is integrated into the core package. If a utility is already present in `astropy.utils`, the package should always use that utility instead of re-implementing it in `packagename.utils` module.

36.2 Documentation and Testing

- Docstrings must be present for all public classes/methods/functions, and must follow the form outlined in the *Writing Documentation* document.
- Write usage examples in the docstrings of all classes and functions whenever possible. These examples should be short and simple to reproduce—users should be able to copy them verbatim and run them. These examples should, whenever possible, be in the *doctest* format and will be executed as part of the test suite.
- Unit tests should be provided for as many public methods and functions as possible, and should adhere to the standards set in the *Testing Guidelines* document.

36.3 Data and Configuration

- Packages can include data in a directory named `data` inside a subpackage source directory as long as it is less than about 100 kb. These data should always be accessed via the `astropy.utils.data.get_pkg_data_fileobj()` or `astropy.utils.data.get_pkg_data_filename()` functions. If the data exceeds this size, it should be hosted outside the source code repository, either at a third-party location on the internet or the astropy data server. In either case, it should always be downloaded using the `astropy.utils.data.get_pkg_data_fileobj()` or `astropy.utils.data.get_pkg_data_filename()` functions. If a specific version of a data file is needed, the hash mechanism described in `astropy.utils.data` should be used.
- All persistent configuration should use the `astropy.config.ConfigurationItem` mechanism. Such configuration items should be placed at the top of the module or package that makes use of them, and supply a description sufficient for users to understand what the setting changes.

36.4 Standard output, warnings, and errors

The built-in `print(...)` function should only be used for output that is explicitly requested by the user, for example `print_header(...)` or `list_catalogs(...)`. Any other standard output, warnings, and errors should follow these rules:

- For errors/exceptions, one should always use `raise` with one of the built-in exception classes, or a custom exception class. The nondescript `Exception` class should be avoided as much as possible, in favor of more specific exceptions (`IOError`, `ValueError`, etc.).
- For warnings, one should always use `warnings.warn(message, warning_class)`. These get redirected to `log.warn` by default, but one can still use the standard warning-catching mechanism and custom warning classes. The warning class should be either `AstropyUserWarning` or inherit from it.
- For informational and debugging messages, one should always use `log.info(message)` and `log.debug(message)`.

The logging system uses the built-in Python `logging` module. The logger can be imported using:

```
from astropy import log
```

36.5 Coding Style/Conventions

- The code will follow the standard [PEP8 Style Guide for Python Code](#). In particular, this includes using only 4 spaces for indentation, and never tabs.
- One exception is to be made from the PEP8 style: new style relative imports of the form `from . import modname` are allowed and required for Astropy, as opposed to absolute (as PEP8 suggests) or the simpler `import modname` syntax. This is primarily due to improved relative import support since PEP8 was developed, and to simplify the process of moving modules.

Note: There are multiple options for testing PEP8 compliance of code, see [Testing Guidelines](#) for more information. See [Emacs setup for following coding guidelines](#) for some configuration options for Emacs that helps in ensuring conformance to PEP8.

- Astropy source code should contain a comment at the beginning of the file (or immediately after the `#!/usr/bin env python` command, if relevant) pointing to the license for the Astropy source code. This line should say:


```
# Licensed under a 3-clause BSD style license - see LICENSE.rst
```
- The `import numpy as np`, `import matplotlib as mpl`, and `import matplotlib.pyplot as plt` naming conventions should be used wherever relevant. `from packagename import *` should never be used, except as a tool to flatten the namespace of a module. An example of the allowed usage is given in [Acceptable use of from module import *](#).
- Classes should either use direct variable access, or python's property mechanism for setting object instance variables. `get_value/set_value` style methods should be used only when getting and setting the values requires a computationally-expensive operation. [Properties vs. get/set](#) below illustrates this guideline.
- All new classes should be new-style classes inheriting from `object` (in Python 3 this is a non-issue as all classes are new-style by default). The one exception to this rule is older classes in third-party libraries such the Python standard library or `numpy`.

- Classes should use the builtin `super()` function when making calls to methods in their super-class(es) unless there are specific reasons not to. `super()` should be used consistently in all subclasses since it does not work otherwise. *super() vs. Direct Calling* illustrates why this is important.
- Multiple inheritance should be avoided in general without good reason. Multiple inheritance is complicated to implement well, which is why many object-oriented languages, like Java, do not allow it at all. Python does enable multiple inheritance through use of the [C3 Linearization](#) algorithm, which provides a consistent method resolution ordering. Non-trivial multiple-inheritance schemes should not be attempted without good justification, or without understanding how C3 is used to determine method resolution order. However, trivial multiple inheritance using orthogonal base classes, known as the ‘mixin’ pattern, may be used.
- `__init__.py` files for modules should not contain any significant implementation code. `__init__.py` can contain docstrings and code for organizing the module layout, however (e.g. `from submodule import *` in accord with the guideline above). If a module is small enough that it fits in one file, it should simply be a single file, rather than a directory with an `__init__.py` file.
- When `try...except` blocks are used to catch exceptions, the `as` syntax should always be used, because this is available in all supported versions of python and is less ambiguous syntax (see *try...except block “as” syntax*).
- Command-line scripts should follow the form outlined in the *Writing Command-Line Scripts* document.

36.6 Unicode guidelines

For maximum compatibility, we need to assume that writing non-ascii characters to the console or to files will not work. However, for those that have a correctly configured Unicode environment, we should allow them to opt-in to take advantage of Unicode output when appropriate. Therefore, there is a global configuration option, `astropy.conf.unicode_output` to enable Unicode output of values, set to `False` by default.

The following conventions should be used for classes that define the standard string conversion methods (`__str__`, `__repr__`, `__unicode__`, `__bytes__`, and `__format__`). In the bullets below, the phrase “unicode instance” is used to refer to `unicode` on Python 2 and `str` on Python 3. The phrase “bytes instance” is used to refer to `str` on Python 2 and `bytes` on Python 3.

- `__repr__`: Return a “unicode instance” (for historical reasons, could also be a “bytes instance” on Python 2, though not preferred) containing only 7-bit characters.
- `__str__` on Python 2 / `__bytes__` on Python 3: Return a “bytes instance” containing only 7-bit characters.
- `__unicode__` on Python 2 / `__str__` on Python 3: Return a “unicode instance”. If `astropy.conf.unicode_output` is `False`, it must contain only 7-bit characters. If `astropy.conf.unicode_output` is `True`, it may contain non-ascii characters when applicable.
- `__format__`: Return a “unicode instance”. If `astropy.UNICODE_OUTPUT` is `False`, it must contain only 7-bit characters. If `astropy.conf.unicode_output` is `True`, it may contain non-ascii characters when applicable.

For classes that are expected to roundtrip through strings (unicode or bytes), the parser must accept either the output of `__str__` or `__unicode__` unambiguously. Additionally, `__repr__` should roundtrip when that makes sense.

This design generally follows Postel’s Law: “Be liberal in what you accept, and conservative in what you send”.

The following example class shows a way to implement this (using `six` for Python 2 and 3 cross-version compatibility):

```
# -*- coding: utf-8 -*-  
  
from __future__ import unicode_literals  
  
from astropy.extern import six  
from astropy import conf
```

```

class FloatList(object):
    def __init__(self, init):
        if isinstance(init, six.text_type):
            init = init.split('||')
        elif isinstance(init, bytes):
            init = init.split(b'||')
        self.x = [float(x) for x in init]

    def __repr__(self):
        # Return unicode object containing no non-ascii characters
        return '<FloatList [{0}]>'.format(', '.join(
            six.text_type(x) for x in self.x))

    def __bytes__(self):
        return b'||'.join(bytes(x) for x in self.x)
if six.PY2:
    __str__ = __bytes__

    def __unicode__(self):
        if astropy.conf.unicode_output:
            return '||'.join(six.text_type(x) for x in self.x)
        else:
            return self.__bytes__().decode('ascii')
if six.PY3:
    __str__ = __unicode__

```

Additionally, there is a test helper, `astropy.test.helper.assert_follows_unicode_guidelines` to ensure that a class follows the Unicode guidelines outlined above. The following example test will test that our example class above is compliant:

```

def test_unicode_guidelines():
    from astropy.test.helper import assert_follows_unicode_guidelines
    assert_follows_unicode_guidelines(FloatList(b'5|4|3|2'), roundtrip=True)

```

36.7 Including C Code

- C extensions are only allowed when they provide a significant performance enhancement over pure python, or a robust C library already exists to provided the needed functionality. When C extensions are used, the Python interface must meet the aforementioned python interface guidelines.
- The use of [Cython](#) is strongly recommended for C extensions, as per the example in the template package. [Cython](#) extensions should store `.pyx` files in the source code repository, but they should be compiled to `.c` files that are updated in the repository when important changes are made to the `.pyx` file.
- If a C extension has a dependency on an external C library, the source code for the library should be bundled with the Astropy core, provided the license for the C library is compatible with the Astropy license. Additionally, the package must be compatible with using a system-installed library in place of the library included in Astropy.
- In cases where C extensions are needed but [Cython](#) cannot be used, the [PEP 7 Style Guide for C Code](#) is recommended.
- C extensions ([Cython](#) or otherwise) should provide the necessary information for building the extension via the mechanisms described in [C or Cython Extensions](#).

36.8 Writing portable code for Python 2 and 3

As of astropy 0.3, the `six` library is included to allow supporting Python 2 and 3 from a single code base. The use of the `2to3` tool is being phased out in favor of using `six`.

To start using `six` instead of `2to3` in a package, you first need to put the following in the package's `setup_package.py` file:

```
def requires_2to3():
    return False
```

This section is mainly about moving existing code that works with `2to3` to using `six`. It is not a complete guide to Python 2 and 3 compatibility.

36.8.1 Welcome to the `__future__`

The top of every `.py` file should include the following:

```
from __future__ import (absolute_import, division, print_function,
                        unicode_literals)
```

This will make the Python 2 interpreter behave as close to Python 3 as possible.

All files should also import `six`, whether they are using it or not, just to make moving code between modules easier, as `six` gets used *a lot*:

```
from ..extern import six
```

(where `extern` refers to `astropy.extern`). Do not import `six` from the top-level: only import the copy of `six` included with astropy.

36.8.2 Finding places to use `six`

Unfortunately, the only way to be certain that code works on both Python 2 and 3 is to make sure it is covered by unit tests.

However, the `2to3` commandline tool can also be used to locate places that require special handling with `six`. Starting from Python 2 code, or code that is known to work on both Python 2 and 3 by processing it with the `2to3` tool (which is most of the existing code in astropy), simply run `2to3` on the file to display the changes it would make in diff format. This diff can be used to highlight places that need to be updated to use `six`.

For example, most things that have been renamed between Python 2 and 3 are in the `six.moves` namespace, so given this Python 2 code:

```
import cPickle
```

it can be replaced with:

```
from ..extern.six.moves import cPickle
```

Note: The `modernize` tool aims to convert Python 2 code to portable code that uses `six`, but at the time of this writing, it is not feature complete and misses many important transformations.

The `six` documentation serves as a good reference for the sorts of things that need to be updated.

36.8.3 Not so fast on that Unicode thing

By importing `unicode_literals` from `__future__`, many things that were once byte strings on Python 2 will now be unicode strings. This is mostly a good thing, as the behavior of the code will be more consistent between Python 2 and 3. However, certain third-party libraries still assume certain values will be byte strings on Python 2.

For example, when specifying Numpy structured dtypes, all strings must be byte strings on Python 2 and unicode strings on Python 3. The easiest way to handle this is to force cast them using `str()`, for example:

```
x = np.array([1.0, 2.0, 3.0], dtype=[(str('name'), '>f8')])
```

The same is true of structure specifiers in the built-in `struct` module on Python 2.6.

`pytest.mark.skipif` also requires a “native” string, i.e.:

```
@pytest.mark.skipif(str('CONDITIONAL'))
```

36.8.4 Iteration

The behavior of the methods for iterating over the items, values and keys of a dictionary has changed in Python 3. Additionally, other built-in functions such as `zip`, `range` and `map` have changed to return iterators rather than temporary lists.

In many cases, the performance implications of iterating vs. creating a temporary list won’t matter, so it’s tempting to use the form that is simplest to read. However, that results in code that behaves differently on Python 2 and 3, leading to subtle bugs that may not be detected by the regression tests. Therefore, unless the loop in question is provably simple and doesn’t call into other code, the `six` versions that ensure the same behavior on both Python 2 and 3 should be used. The following table shows the mapping of equivalent semantics between Python 2, 3 and `six` for `dict.items()`:

Python 2	Python 3	six
<code>d.items()</code>	<code>list(d.items())</code>	<code>list(six.iteritems(d))</code>
<code>d.iteritems()</code>	<code>d.items()</code>	<code>six.iteritems(d)</code>

The above table holds true, analogously, for values, keys, `zip`, `range` and `map`.

Note that for keys only, `list(d)` is an acceptable shortcut to `list(six.iterkeys(d))`.

36.8.5 Issues with `\u` escapes

When `from __future__ import unicode_literals` is used, all string literals (not preceded with a ‘b’) will become unicode literals.

Normally, one would use “raw” string literals to encode strings that contain a lot of slashes that we don’t want Python to interpret as special characters. Unfortunately, on Python 2, there is no way to represent ‘`\u`’ in a raw unicode string literal, since it will always be interpreted as the start of a unicode character escape, such as ‘`\u20af`’. The only solution is to use a regular (non-raw) string literal and repeat all slashes, e.g. “`\\usepackage{foo}`”.

The following shows the problem on Python 2:

```
>>> ur'\u'
File "<stdin>", line 1
SyntaxError: (unicode error) 'rawunicodeescape' codec can't decode bytes in
position 0-1: truncated \uXXXX
>>> ur'\\u'
u'\\\\u'
>>> u'\u'
File "<stdin>", line 1
SyntaxError: (unicode error) 'unicodeescape' codec can't decode bytes in
```

```
position 0-1: truncated \uXXXX escape
>>> u'\u'
u'\u'
  File "<stdin>", line 1
SyntaxError: (unicode error) 'unicodeescape' codec can't decode bytes in
```

This bug has been fixed in Python 3, however, we can't take advantage of that and still support Python 2:

```
>>> r'\u'
'\u'
>>> r'\u'
'\u'
>>> r'\u'
'\u'
>>> '\u'
  File "<stdin>", line 1
SyntaxError: (unicode error) 'unicodeescape' codec can't decode bytes in
position 0-1: truncated \uXXXX escape
>>> '\u'
'\u'
  File "<stdin>", line 1
SyntaxError: (unicode error) 'unicodeescape' codec can't decode bytes in
```

36.9 Requirements Specific to Affiliated Packages

- Affiliated packages implementing many classes/functions not relevant to the affiliated package itself (for example leftover code from a previous package) will not be accepted - the package should only include the required functionality and relevant extensions.
- Affiliated packages are required to follow the layout and documentation form of the template package included in the core package source distribution.
- Affiliated packages must be registered on the [Python Package Index](#), with proper metadata for downloading and installing the source package.
- The `astropy` root package name should not be used by affiliated packages - it is reserved for use by the core package. Recommended naming conventions for an affiliated package are either simply `packagename` or `awastropy.packagename` (“affiliated with Astropy”).

36.10 Examples

This section shows a few examples (not all of which are correct!) to illustrate points from the guidelines. These will be moved into the template project once it has been written.

36.10.1 Properties vs. `get_/set_`

This example shows a sample class illustrating the guideline regarding the use of properties as opposed to getter/setter methods.

Let's assuming you've defined a `'class: `Star`'` class and create an instance like this:

```
>>> s = Star(B=5.48, V=4.83)
```

You should always use attribute syntax like this:

```
>>> s.color = 0.4
>>> print(s.color)
0.4
```

Rather than like this:

```
>>> s.set_color(0.4) #Bad form!
>>> print(s.get_color()) #Bad form!
0.4
```

Using python properties, attribute syntax can still do anything possible with a get/set method. For lengthy or complex calculations, however, use a method:

```
>>> print(s.compute_color(5800, age=5e9))
0.4
```

36.10.2 super() vs. Direct Calling

This example shows why the use of `super()` leads to a more consistent method resolution order than manually calling methods of the super classes in a multiple inheritance case:

This is dangerous and bug-prone!

```
class A(object):
    def method(self):
        print('Doing A')

class B(A):
    def method(self):
        print('Doing B')
        A.method(self)

class C(A):
    def method(self):
        print('Doing C')
        A.method(self)

class D(C, B):
    def method(self):
        print('Doing D')
        C.method(self)
        B.method(self)
```

if you then do:

```
>>> b = B()
>>> b.method()
```

you will see:

```
Doing B
Doing A
```

which is what you expect, and similarly for C. However, if you do:

```
>>> d = D()
>>> d.method()
```

you might expect to see the methods called in the order D, B, C, A but instead you see:

```
Doing D
Doing C
Doing A
Doing B
Doing A
```

because both `B.method()` and `C.method()` call `A.method()` unaware of the fact that they're being called as part of a chain in a hierarchy. When `C.method()` is called it is unaware that it's being called from a subclass that inherits from both B and C, and that `B.method()` should be called next. By calling `super()` the entire method resolution order for D is precomputed, enabling each superclass to cooperatively determine which class should be handed control in the next `super()` call:

```
# This is safer

class A(object):
    def method(self):
        print('Doing A')

class B(A):
    def method(self):
        print('Doing B')
        super(B, self).method()

class C(A):
    def method(self):
        print('Doing C')
        super(C, self).method()

class D(C, B):
    def method(self):
        print('Doing D')
        super(D, self).method()
```

```
>>> d = D()
>>> d.method()
Doing D
Doing C
Doing B
Doing A
```

As you can see, each superclass's method is entered only once. For this to work it is very important that each method in a class that calls its superclass's version of that method use `super()` instead of calling the method directly. In the most common case of single-inheritance, using `super()` is functionally equivalent to calling the superclass's method directly. But as soon as a class is used in a multiple-inheritance hierarchy it must use `super()` in order to cooperate with other classes in the hierarchy.

Note: For more information on the the benefits of `super()`, see <http://rhettinger.wordpress.com/2011/05/26/super-considered-super/>

36.10.3 Acceptable use of `from module import *`

`from module import *` is discouraged in a module that contains implementation code, as it impedes clarity and often imports unused variables. It can, however, be used for a package that is laid out in the following manner:

```
packagename
packagename/__init__.py
packagename/submodule1.py
packagename/submodule2.py
```

In this case, `packagename/__init__.py` may be:

```
"""
A docstring describing the package goes here
"""
from submodule1 import *
from submodule2 import *
```

This allows functions or classes in the submodules to be used directly as `packagename.foo` rather than `packagename.submodule1.foo`. If this is used, it is strongly recommended that the submodules make use of the `__all__` variable to specify which modules should be imported. Thus, `submodule2.py` might read:

```
from numpy import array, linspace

__all__ = ('foo', 'AClass')

def foo(bar):
    #the function would be defined here
    pass

class AClass(object):
    #the class is defined here
    pass
```

This ensures that `from submodule import *` only imports `:func: 'foo'` and `:class: 'AClass'`, but not `:class: 'numpy.array'` or `:func: 'numpy.linspace'`.

36.10.4 `try...except` block “as” syntax

Catching of exceptions should always use this syntax:

```
try:
    ... some code that might produce a variety of exceptions ...
except ImportError as e:
    if 'somemodule' in e.args[0]:
        #for whatever reason, failed import of somemodule is ok
        pass
    else:
        raise
except ValueError, TypeError as e:
    msg = 'Hit an input problem, which is ok,'
    msg2 = 'but we're printing it here just so you know:'
    print msg, msg2, e
```

This avoids the old style syntax of `except ImportError, e` or `except (ValueError, TypeError), e`, which is dangerous because it's easy to instead accidentally do something like `except ValueError, TypeError`, which won't catch `TypeError`.

36.11 Additional Resources

Further tips and hints relating to the coding guidelines are included below.

36.11.1 Emacs setup for following coding guidelines

The Astropy coding guidelines are listed in *Coding Guidelines*. This document will describe some configuration options for Emacs, that will help in ensuring that Python code satisfies the guidelines. Emacs can be configured in several different ways. So instead of providing a drop in configuration file, only the individual configurations are presented below.

For this setup we will need `flymake`, `pyflakes` and the `pep8` Python script, in addition to `python-mode`.

Flymake comes with Emacs 23. The rest can be obtained from their websites, or can be installed using `easy_install` or `pip`.

Global settings

No tabs

This setting will cause all tabs to be replaced with spaces. The number of spaces to use is set in the *Basic settings* section below.

```
;; Don't use TABS for indentations.
(setq-default indent-tabs-mode nil)
```

Maximum number of characters in a line

Emacs will automatically insert a new line after “fill-column” number of columns. PEP8 specifies a maximum of 79, but this can be set to a smaller value also, for example 72.

```
;; Set the number to the number of columns to use.
(setq-default fill-column 79)
```

```
;; Add Autofill mode to mode hooks.
(add-hook 'text-mode-hook 'turn-on-auto-fill)
```

```
;; Show line number in the mode line.
(line-number-mode 1)
```

```
;; Show column number in the mode line.
(column-number-mode 1)
```

Syntax highlighting

Enable syntax highlighting. This will also highlight lines that form a region.

```
(global-font-lock-mode 1)
```

Python specific settings

Basic settings

Indentation is automatically added. When a tab is pressed it is replaced with 4 spaces. When backspace is pressed on an empty line, the cursor will jump to the previous indentation level.

```
(load-library "python")

(auto-load 'python-mode "python-mode" "Python Mode." t)
(add-to-list 'auto-mode-alist '("\\.py\\\\" . python-mode))
(add-to-list 'interpreter-mode-alist '("python" . python-mode))

(setq interpreter-mode-alist
      (cons '("python" . python-mode)
            interpreter-mode-alist)
      python-mode-hook
      '(lambda () (progn
                  (set-variable 'py-indent-offset 4)
                  (set-variable 'indent-tabs-mode nil))))
```

Highlight the column where a line must stop

The “fill-column” column is highlighted in red. For this to work, download `column-marker.el` and place it in the Emacs configuration directory.

```
;; Highlight character at "fill-column" position.
(require 'column-marker)
(set-face-background 'column-marker-1 "red")
(add-hook 'python-mode-hook
          (lambda () (interactive)
              (column-marker-1 fill-column)))
```

Flymake

Flymake will mark lines that do not satisfy syntax requirements in red. When cursor is on such a line a message is displayed in the mini-buffer. When mouse pointer is on such a line a “tool tip” message is also shown.

For flymake to work with `pep8` and `pyflakes`, create an executable file named `pychecker` with the following contents. This file must be in the system path.

```
#!/bin/bash

pyflakes "$1"
pep8 --ignore=E221,E701,E202 --repeat "$1"
true
```

Also download `flymake-cursor.el` and place it in the Emacs configuration directory. Then add the following code to the Emacs configuration:

```
;; Setup for Flymake code checking.
(require 'flymake)
(load-library "flymake-cursor")

;; Script that flymake uses to check code. This script must be
;; present in the system path.
```

```
(setq pycodechecker "pychecker")

(when (load "flymake" t)
  (defun flymake-pycodecheck-init ()
    (let* ((temp-file (flymake-init-create-temp-buffer-copy
                      'flymake-create-temp-inplace))
          (local-file (file-relative-name
                      temp-file
                      (file-name-directory buffer-file-name))))
      (list pycodechecker (list local-file))))
  (add-to-list 'flymake-allowed-file-name-masks
    ('("\\.py\\\\" flymake-pycodecheck-init)))

(add-hook 'python-mode-hook 'flymake-mode)
```

Note: Flymake will save files with suffix `_flymake` in the current directory. If it crashes for some reason, then these files will not get deleted.

Sometimes there is a delay in refreshing the results.

Delete trailing white spaces and blank lines

To manually delete trailing whitespaces, press `C-t C-w`, which will run the command “`delete-whitespaces`”. This command is also run when a file is saved, and hence all trailing whitespaces will be deleted on saving a Python file.

To make sure that all “words” are separated by only one space, type `M-SPC` (use the `ALT` key since `M-SPC` sometimes brings up a context menu.).

To collapse a set of blank lines to one blank line, place the cursor on one of these and press `C-x C-o`. This is useful for deleting multiple blank lines at the end of a file.

```
;; Remove trailing whitespace manually by typing C-t C-w.
(add-hook 'python-mode-hook
  (lambda ()
    (local-set-key (kbd "C-t C-w")
      'delete-trailing-whitespace)))

;; Automatically remove trailing whitespace when file is saved.
(add-hook 'python-mode-hook
  (lambda ()
    (add-hook 'local-write-file-hooks
      '(lambda ()
         (save-excursion
           (delete-trailing-whitespace)))))))

;; Use M-SPC (use ALT key) to make sure that words are separated by
;; just one space. Use C-x C-o to collapse a set of empty lines
;; around the cursor to one empty line. Useful for deleting all but
;; one blank line at end of file. To do this go to end of file (M->)
;; and type C-x C-o.
```

WRITING DOCUMENTATION

High-quality, consistent documentation for astronomy code is one of the major goals of the Astropy project. Hence, we describe our documentation procedures and rules here. For the astropy core project we try to keep to these as closely as possible, while the standards for affiliated packages are somewhat looser. (These procedures and guidelines are still recommended for affiliated packages, as they encourage useful documentation, a characteristic often lacking in professional astronomy software.)

37.1 Building the Documentation from source

For information about building the documentation from source, see the *Building documentation* section in the installation instructions.

37.2 Astropy Documentation Rules and Guidelines

This section describes the standards for documentation format affiliated packages that must follow for consideration of integration into the core module, as well as the standard Astropy docstring format.

- All documentation should be written use the Sphinx documentation tool.
- The template package will provide a recommended general structure for documentation.
- Docstrings must be provided for all public classes, methods, and functions.
- Docstrings will be incorporated into the documentation using a version of numpdoc included with Astropy, and should follow the *Astropy Docstring Rules*.
- Examples and/or tutorials are strongly encouraged for typical use-cases of a particular module or class.
- Any external package dependencies aside from NumPy, SciPy, or Matplotlib must be explicitly mentioned in the documentation.
- Configuration options using the `astropy.config` mechanisms must be explicitly mentioned in the documentation.

The details of the docstring format are described on a separate page:

37.2.1 Astropy Docstring Rules

The original source for these docstring standards is the NumPy project, and the associated numpdoc tools. The most up-to-date version of these standards can be found at [numpy's github site](#). The guidelines below have been adapted to the Astropy package.

Overview

In general, we follow the standard Python style conventions as described here:

- [Style Guide for C Code](#)
- [Style Guide for Python Code](#)
- [Docstring Conventions](#)

Additional PEPs of interest regarding documentation of code:

- [Docstring Processing Framework](#)
- [Docutils Design Specification](#)

Use a code checker:

- [pylint](#)
- [pyflakes](#)
- [pep8.py](#)

The following import conventions are used throughout the Astropy source and documentation:

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
```

Do not abbreviate `scipy`. There is no motivating use case to abbreviate it in the real world, so we avoid it in the documentation to avoid confusion.

It is not necessary to do `import numpy as np` at the beginning of an example. However, some sub-modules, such as `fft`, are not imported by default, and you have to include them explicitly:

```
import numpy.fft
```

after which you may use it:

```
np.fft.fft2(...)
```

Docstring Standard

A documentation string (docstring) is a string that describes a module, function, class, or method definition. The docstring is a special attribute of the object (`object.__doc__`) and, for consistency, is surrounded by triple double quotes, i.e.:

```
"""This is the form of a docstring.

It can be spread over several lines.

"""
```

NumPy and SciPy have defined a common convention for docstrings that provides for consistency, while also allowing our toolchain to produce well-formatted reference guides. This format should be used for Astropy docstrings.

This docstring standard uses [re-structured text \(reST\)](#) syntax and is rendered using [Sphinx](#) (a pre-processor that understands the particular documentation style we are using). While a rich set of markup is available, we limit ourselves to a very basic subset, in order to provide docstrings that are easy to read on text-only terminals.

A guiding principle is that human readers of the text are given precedence over contorting docstrings so our tools produce nice output. Rather than sacrificing the readability of the docstrings, we have written pre-processors to assist `Sphinx` in its task.

The length of docstring lines should be kept to 75 characters to facilitate reading the docstrings in text terminals.

Sections

The sections of the docstring are:

1. Short summary

A one-line summary that does not use variable names or the function name, e.g.

```
def add(a, b):
    """The sum of two numbers.

    """
```

The function signature is normally found by introspection and displayed by the help function. For some functions (notably those written in C) the signature is not available, so we have to specify it as the first line of the docstring:

```
"""
add(a, b)

The sum of two numbers.

"""
```

2. Deprecation warning

A section (use if applicable) to warn users that the object is deprecated. Section contents should include:

- In what Astropy version the object was deprecated, and when it will be removed.
- Reason for deprecation if this is useful information (e.g., object is superseded, duplicates functionality found elsewhere, etc.).
- New recommended way of obtaining the same functionality.

This section should use the `note` Sphinx directive instead of an underlined section header.

```
.. note:: Deprecated in Astropy 1.2
        `ndobj_old` will be removed in Astropy 2.0, it is replaced by
        `ndobj_new` because the latter works also with array subclasses.
```

3. Extended summary

A few sentences giving an extended description. This section should be used to clarify *functionality*, not to discuss implementation detail or background theory, which should rather be explored in the **notes** section below. You may refer to the parameters and the function name, but parameter descriptions still belong in the **parameters** section.

4. Parameters

Description of the function arguments, keywords and their respective types.

```
Parameters
-----
x : type
    Description of parameter `x`.
```

Enclose variables in single backticks.

For the parameter types, be as precise as possible. Below are a few examples of parameters and their types.

```
Parameters
-----
filename : str
copy : bool
dtype : data-type
iterable : iterable object
shape : int or tuple of int
files : list of str
```

If it is not necessary to specify a keyword argument, use `optional`:

```
x : int, optional
```

Optional keyword parameters have default values, which are displayed as part of the function signature. They can also be detailed in the description:

```
Description of parameter `x` (the default is -1, which implies summation
over all axes).
```

When a parameter can only assume one of a fixed set of values, those values can be listed in braces:

```
order : {'C', 'F', 'A'}
    Description of `order`.
```

When two or more input parameters have exactly the same type, shape and description, they can be combined:

```
x1, x2 : array_like
    Input arrays, description of `x1`, `x2`.
```

5. Returns

Explanation of the returned values and their types, of the same format as **parameters**.

6. Other parameters

An optional section used to describe infrequently used parameters. It should only be used if a function has a large number of keyword parameters, to prevent cluttering the **parameters** section.

7. Raises

An optional section detailing which errors get raised and under what conditions:

```
Raises
-----
InvalidWCSEException
    If the WCS information is invalid.
```

This section should be used judiciously, i.e only for errors that are non-obvious or have a large chance of getting raised.

8. See Also

An optional section used to refer to related code. This section can be very useful, but should be used judiciously. The goal is to direct users to other functions they may not be aware of, or have easy means of discovering (by looking at the module docstring, for example). Routines whose docstrings further explain parameters used by this function are good candidates.

As an example, for a hypothetical function `astropy.wcs.world2pix` converting sky to pixel coordinates, we would have:

See Also

`pix2world` : Convert pixel to sky coordinates

When referring to functions in the same sub-module, no prefix is needed, and the tree is searched upwards for a match.

Prefix functions from other sub-modules appropriately. E.g., whilst documenting a hypothetical `astropy.vo` module, refer to a function in `table` by

`table.read` : Read in a VO table

When referring to an entirely different module:

`astropy.coords` : Coordinate handling routines

Functions may be listed without descriptions, and this is preferable if the functionality is clear from the function name:

See Also

`func_a` : Function a with its description.

`func_b`, `func_c`, `func_d`

`func_e`

9. Notes

An optional section that provides additional information about the code, possibly including a discussion of the algorithm. This section may include mathematical equations, written in [LaTeX](#) format:

The FFT is a fast implementation of the discrete Fourier transform:

```
.. math:: X(e^{j\omega} ) = x(n)e^{ - j\omega n}
```

Equations can also be typeset underneath the `math` directive:

The discrete-time Fourier time-convolution property states that

```
.. math::
```

```
    x(n) * y(n) \Leftrightarrow X(e^{j\omega} )Y(e^{j\omega} )\ \
    another equation here
```

`Math` can furthermore be used inline, i.e.

The value of `:math:`\omega`` is larger than 5.

Variable names are displayed in typewriter font, obtained by using `\mathtt{var}`:

We square the input parameter `\alpha` to obtain

```
:math:`\mathtt{alpha}^2`.
```

Note that [LaTeX](#) is not particularly easy to read, so use equations sparingly.

Images are allowed, but should not be central to the explanation; users viewing the docstring as text must be able to comprehend its meaning without resorting to an image viewer. These additional illustrations are included using:

```
.. image:: filename
```

where `filename` is a path relative to the reference guide source directory.

10. References

References cited in the **notes** section may be listed here, e.g. if you cited the article below using the text [1]_, include it as in the list as follows:

```
.. [1] O. McNoleg, "The integration of GIS, remote sensing,
    expert systems and adaptive co-kriging for environmental habitat
    modelling of the Highland Haggis using object-oriented, fuzzy-logic
    and neural-network techniques," Computers & Geosciences, vol. 22,
    pp. 585-588, 1996.
```

which renders as

Referencing sources of a temporary nature, like web pages, is discouraged. References are meant to augment the docstring, but should not be required to understand it. References are numbered, starting from one, in the order in which they are cited.

11. Examples

An optional section for examples, using the `doctest` format. This section is meant to illustrate usage, not to provide a testing framework – for that, use the `tests/` directory. While optional, this section is very strongly encouraged.

When multiple examples are provided, they should be separated by blank lines. Comments explaining the examples should have blank lines both above and below them:

```
>>> astropy.wcs.world2pix(233.2, -12.3)
(134.5, 233.1)
```

Comment explaining the second example

```
>>> astropy.coords.fk5_to_gal("00:42:44.33 +41:16:07.5")
(121.1743, -21.5733)
```

For tests with a result that is random or platform-dependent, mark the output as such:

```
>>> astropy.coords.randomize_position(244.9, 44.2, radius=0.1)
(244.855, 44.13) # random
```

It is not necessary to use the `doctest` markup `<BLANKLINE>` to indicate empty lines in the output. The examples may assume that `import numpy as np` is executed before the example code.

Documenting classes

Class docstrings

Use the same sections as outlined above (all except `Returns` are applicable). The constructor (`__init__`) should also be documented here, the `Parameters` section of the docstring details the constructors parameters.

An `Attributes` section, located below the `Parameters` section, may be used to describe class variables:

```
Attributes
-----
x : float
    The X coordinate.
y : float
    The Y coordinate.
```

Attributes that are properties and have their own docstrings can be simply listed by name:

```

Attributes
-----
real
imag
x : float
    The X coordinate
y : float
    The Y coordinate

```

In general, it is not necessary to list class methods. Those that are not part of the public API have names that start with an underscore. In some cases, however, a class may have a great many methods, of which only a few are relevant (e.g., subclasses of `ndarray`). Then, it becomes useful to have an additional `Methods` section:

```

class Table(ndarray):
    """
    A class to represent tables of data

    ...

    Attributes
    -----
    columns : list
        List of columns

    Methods
    -----
    read(filename)
        Read a table from a file
    sort(column, order='ascending')
        Sort by `column`
    """

```

If it is necessary to explain a private method (use with care!), it can be referred to in the **extended summary** or the **notes**. Do not list private methods in the `Methods` section.

Do not list `self` as the first parameter of a method.

Method docstrings

Document these as you would any other function. Do not include `self` in the list of parameters. If a method has an equivalent function, the function docstring should contain the detailed documentation, and the method docstring should refer to it. Only put brief `Summary` and `See Also` sections in the method docstring.

Documenting class instances

Instances of classes that are part of the Astropy API may require some care. To give these instances a useful docstring, we do the following:

- **Single instance:** If only a single instance of a class is exposed, document the class. Examples can use the instance name.
- **Multiple instances:** If multiple instances are exposed, docstrings for each instance are written and assigned to the instances' `__doc__` attributes at run time. The class is documented as usual, and the exposed instances can be mentioned in the `Notes` and `See Also` sections.

Documenting constants

Use the same sections as outlined for functions where applicable:

1. summary
2. extended summary (optional)
3. see also (optional)
4. references (optional)
5. examples (optional)

Docstrings for constants will not be visible in text terminals (constants are of immutable type, so docstrings can not be assigned to them like for for class instances), but will appear in the documentation built with Sphinx.

Documenting modules

Each module should have a docstring with at least a summary line. Other sections are optional, and should be used in the same order as for documenting functions when they are appropriate:

1. summary
2. extended summary
3. routine listings
4. see also
5. notes
6. references
7. examples

Routine listings are encouraged, especially for large modules, for which it is hard to get a good overview of all functionality provided by looking at the source file(s) or the `__all__` dict.

Note that license and author info, while often included in source files, do not belong in docstrings.

Other points to keep in mind

- Notes and Warnings : If there are points in the docstring that deserve special emphasis, the reST directives for a note or warning can be used in the vicinity of the context of the warning (inside a section). Syntax:

```
.. warning:: Warning text.  
  
.. note:: Note text.
```

Use these sparingly, as they do not look very good in text terminals and are not often necessary. One situation in which a warning can be useful is for marking a known bug that is not yet fixed.

- Questions and Answers : For general questions on how to write docstrings that are not answered in this document, refer to <http://docs.scipy.org/numPy/Questions+Answers/>.
- `array_like` : For functions that take arguments which can have not only a type `ndarray`, but also types that can be converted to an `ndarray` (i.e. scalar types, sequence types), those arguments can be documented with type `array_like`.

Common reST concepts

For paragraphs, indentation is significant and indicates indentation in the output. New paragraphs are marked with a blank line.

Use *italics*, **bold**, and `courier` if needed in any explanations (but not for variable names and doctest code or multi-line code). Variable, module and class names should be written between single back-ticks (``astropy``).

A more extensive example of reST markup can be found in [this example document](#); the [quick reference](#) is useful while editing.

Line spacing and indentation are significant and should be carefully followed.

Conclusion

An [example](#) of the format shown here is available. Refer to [How to Build API/Reference Documentation](#) on how to use [Sphinx](#) to build the manual.

37.3 Sphinx Documentation Themes

A custom Sphinx HTML theme is included in the [astropy-helpers](#) package (it is also included in Astropy's source package, but this will be removed after v0.4 and subsequently only be available through [astropy-helpers](#)). This allows the theme to be used by both Astropy and affiliated packages. This is done by setting the theme in the global Astropy sphinx configuration, which is imported in the sphinx configuration of both Astropy and affiliated packages.

37.3.1 Using a different theme for `astropy` or affiliated packages

A different theme can be used by overriding a few sphinx configuration variables set in the global configuration.

- To use a different theme, set `'html_theme'` to the name of a desired builtin Sphinx theme or a custom theme in `package-name/docs/conf.py` (where `'package-name'` is “astropy” or the name of the affiliated package).
- To use a custom theme, additionally: place the theme in `package-name/docs/_themes` and add `'_themes'` to the `'html_theme_path'` variable. See the [Sphinx](#) documentation for more details on theming.

37.3.2 Adding more custom themes to astropy

Additional custom themes can be included in the astropy source tree by placing them in the directory `astropy/astropy/sphinx/themes`, and editing `astropy/astropy/sphinx/setup_package.py` to include the theme (so that it is installed).

37.4 Sphinx extensions

[Astropy-helpers](#) includes a number of sphinx extensions that are used in Astropy and its affiliated packages to facilitate easily documenting code in a homogeneous and readable way.

Note: These extensions are also included with Astropy itself in v0.4 and below, to facilitate backwards-compatibility for existing affiliated packages. The versions actually in `astropy` will not receive further updates, however, and will likely be removed in a future version. So we strongly recommend using the `astropy-helper` versions instead.

37.4.1 `automodapi` Extension

This sphinx extension adds a tools to simplify generating the API documentation for Astropy packages and affiliated packages.

automodapi directive

This directive takes a single argument that must be a module or package. It will produce a block of documentation that includes the docstring for the package, an *automodsumm directive* directive, and an *automod-diagram directive* if there are any classes in the module. If only the main docstring of the module/package is desired in the documentation, use `automodule` instead of `automodapi`.

It accepts the following options:

- **:no-inheritance-diagram:**
If present, the inheritance diagram will not be shown even if the module/package has classes.
- **:skip: str**
This option results in the specified object being skipped, that is the object will *not* be included in the generated documentation. This option may appear any number of times to skip multiple objects.
- **:no-main-docstr:**
If present, the docstring for the module/package will not be generated. The function and class tables will still be used, however.
- **:headings: str**
Specifies the characters (in one string) used as the heading levels used for the generated section. This must have at least 2 characters (any after 2 will be ignored). This also *must* match the rest of the documentation on this page for sphinx to be happy. Defaults to “-^”, which matches the convention used for Python’s documentation, assuming the automodapi call is inside a top-level section (which usually uses ‘=’).
- **:no-heading:**
If specified do not create a top level heading for the section. That is, do not create a title heading with text like “packagename Package”. The actual docstring for the package/module will still be shown, though, unless `:no-main-docstr:` is given.
- **:allowed-package-names: str**
Specifies the packages that functions/classes documented here are allowed to be from, as comma-separated list of package names. If not given, only objects that are actually in a subpackage of the package currently being documented are included.

This extension also adds two sphinx configuration options:

- **automodapi_toctreedirnm**
This must be a string that specifies the name of the directory the automodsumm generated documentation ends up in. This directory path should be relative to the documentation root (e.g., same place as `index.rst`). Defaults to ‘api’.
- **automodapi_writereprocessed**
Should be a bool, and if `True`, will cause `automodapi` to write files with any `automodapi` sections replaced with the content Sphinx processes after `automodapi` has run. The output files are not actually used by sphinx, so this option is only for figuring out the cause of sphinx warnings or other debugging. Defaults to `False`.

37.4.2 automodsumm Extension

This sphinx extension adds two directives for summarizing the public members of a module or package.

These directives are primarily for use with the `automodapi` extension, but can be used independently.

automodsumm directive

This directive will produce an “autosummary”-style table for public attributes of a specified module. See the [sphinx.ext.autosummary](#) extension for details on this process. The main difference from the `autosummary` directive is that `automodsumm` requires manually inputting all attributes that appear in the table, while this captures the entries automatically.

This directive requires a single argument that must be a module or package.

It also accepts any options supported by the `autosummary` directive- see [sphinx.ext.autosummary](#) for details. It also accepts two additional options:

- **`:classes-only:`**
If present, the autosummary table will only contain entries for classes. This cannot be used at the same time with `:functions-only:`.
- **`:functions-only:`**
If present, the autosummary table will only contain entries for functions. This cannot be used at the same time with `:classes-only:`.
- **`:skip: obj1, [obj2, obj3, ...]`**
If present, specifies that the listed objects should be skipped and not have their documentation generated, nor be included in the summary table.
- **`:allowed-package-names: pkgormod1, [pkgormod2, pkgormod3, ...]`**
Specifies the packages that functions/classes documented here are allowed to be from, as comma-separated list of package names. If not given, only objects that are actually in a subpackage of the package currently being documented are included.

This extension also adds one sphinx configuration option:

- **`automodsumm_writereprocessed`**
Should be a bool, and if True, will cause `automodsumm` to write files with any `automodsumm` sections replaced with the content Sphinx processes after `automodsumm` has run. The output files are not actually used by sphinx, so this option is only for figuring out the cause of sphinx warnings or other debugging. Defaults to `False`.

automod-diagram directive

This directive will produce an inheritance diagram like that of the [sphinx.ext.inheritance_diagram](#) extension.

This directive requires a single argument that must be a module or package. It accepts no options.

Note: Like ‘inheritance-diagram’, ‘automod-diagram’ requires [graphviz](#) to generate the inheritance diagram.

37.4.3 edit_on_github Extension

This extension makes it easy to edit documentation on github.

It adds links associated with each docstring that go to the corresponding view source page on Github. From there, the user can push the “Edit” button, edit the docstring, and submit a pull request.

It has the following configuration options (to be set in the project’s `conf.py`):

- **`edit_on_github_project`**
The name of the github project, in the form “username/projectname”.

- **edit_on_github_branch**
The name of the branch to edit. If this is a released version, this should be a git tag referring to that version. For a dev version, it often makes sense for it to be “master”. It may also be a git hash.
- **edit_on_github_source_root**
The location within the source tree of the root of the Python package. Defaults to “lib”.
- **edit_on_github_doc_root**
The location within the source tree of the root of the documentation source. Defaults to “doc”, but it may make sense to set it to “doc/source” if the project uses a separate source directory.
- **edit_on_github_docstring_message**
The phrase displayed in the links to edit a docstring. Defaults to “[edit on github]”.
- **edit_on_github_page_message**
The phrase displayed in the links to edit a RST page. Defaults to “[edit this page on github]”.
- **edit_on_github_help_message**
The phrase displayed as a tooltip on the edit links. Defaults to “Push the Edit button on the next page”
- **edit_on_github_skip_regex**
When the path to the .rst file matches this regular expression, no “edit this page on github” link will be added. Defaults to “_.*”.

37.4.4 numpydoc Extension

This extension (and some related extensions) are a part of the [numpydoc](#) extension written by the NumPy and SciPy, projects, with some tweaks for Astropy. Its main purposes is to reprocess docstrings from code into a form sphinx understands. Generally, there’s no need to interact with it directly, as docstrings following the *Astropy Docstring Rules* will be processed automatically.

37.4.5 Other Extensions

`astropy_helpers.sphinx.ext` includes a few other extensions that are primarily helpers for the other extensions or workarounds for undesired behavior. Their APIs are not included here because we may change them in the future.

TESTING GUIDELINES

This section describes the testing framework and format standards for tests in Astropy core packages (this also serves as recommendations for affiliated packages).

38.1 Testing Framework

The testing framework used by Astropy is the `py.test` framework.

38.2 Running Tests

There are currently three different ways to invoke Astropy tests. Each method invokes `py.test` to run the tests but offers different options when calling.

In addition to running the Astropy tests, these methods can also be called so that they check Python source code for [PEP8 compliance](#). All of the PEP8 testing options require the `pytest-pep8` plugin, which must be installed separately.

38.2.1 `setup.py` test

The safest way to run the astropy test suite is via the `setup` command `test`. This is invoked by running `python setup.py test` while in the astropy source code directory. Run `python setup.py test --help` to see the options to the test command.

Turn on PEP8 checking by passing `--pep8` to the `test` command. This will turn off regular testing and enable PEP8 testing.

Note: This method of running the tests defaults to the version of `py.test` that is bundled with Astropy. To use the locally-installed version, you can set the `ASTROPY_USE_SYSTEM_PYTEST` environment variable, eg.:

```
> ASTROPY_USE_SYSTEM_PYTEST=1 python setup.py test
```

38.2.2 `py.test`

An alternative way to run tests from the command line is to switch to the source code directory of astropy and simply type:

```
py.test
```

`py.test` will look for files that [look like tests](#) in the current directory and all recursive directories then run all the code that [looks like tests](#) within those files.

Note: To test any compiled C/Cython extensions, you must run `python setup.py develop` prior to running the `py.test` command-line script. Otherwise, any tests that make use of these extensions will not succeed. Similarly, in python 3, these tests will not run correctly in the source code, because they need the [2to3](#) tool to be run on them.

You may specify a specific test file or directory at the command line:

```
py.test test_file.py
```

To run a specific test within a file use the `-k` option:

```
py.test test_file.py -k "test_function"
```

You may also use the `-k` option to not run tests by putting a `-` in front of the matching string:

```
py.test test_file.py -k "-test_function"
```

`py.test` has a number of [command line usage options](#).

Turn on PEP8 testing by adding the `--pep8` flag to the `py.test` call. By default regular tests will also be run but these can be turned off by adding `-k pep8`:

```
py.test some_dir --pep8 -k pep8
```

Note: This method of running the tests uses the locally-installed version of `py.test` rather than the bundled one, and hence will fail if the local version it is not up-to-date enough ([py.test 2.2](#) as of this writing).

38.2.3 `astropy.test()`

AstroPy includes a standalone version of `py.test` that allows tests to be run even if `py.test` is not installed. Tests can be run from within AstroPy with:

```
import astropy
astropy.test()
```

This will run all the default tests for AstroPy.

Tests for a specific package can be run by specifying the package in the call to the `test()` function:

```
astropy.test('io.fits')
```

This method works only with package names that can be mapped to AstroPy directories. As an alternative you can test a specific directory or file with the `test_path` option:

```
astropy.test(test_path='wcs/tests/test_wcs.py')
```

The `test_path` must be specified either relative to the working directory or absolutely.

By default `astropy.test()` will skip tests which retrieve data from the internet. To turn these tests on use the `remote_data` flag:

```
astropy.test('io.fits', remote_data=True)
```

In addition, the `test` function supports any of the options that can be passed to `pytest.main()`, and convenience options `verbose=` and `pastebin=`.

Enable PEP8 compliance testing with `pep8=True` in the call to `astropy.test`. This will enable PEP8 checking and disable regular tests.

Note: This method of running the tests defaults to the version of `py.test` that is bundled with Astropy. To use the locally-installed version, you should set the `ASTROPY_USE_SYSTEM_PYTEST` environment variable (see [Configuration system \(`astropy.config`\)](#)) or the `py.test` method described above.

38.2.4 Tox

Tox is a sort of meta-test runner for Python. It installs a project into one or more virtualenvs (usually one for each Python version supported), build and installs the project into each virtualenv, and runs the projects tests (or any other build processes one might want to test). This is a good way to run the tests against multiple installed Python versions locally without pushing to a continuous integration system.

Tox works by detecting the presence of a file called `tox.ini` in the root of a Python project and using that to configure the desired virtualenvs and start the tests. So to run the Astropy tests on multiple Python versions using tox, simply install Tox:

```
$ pip install tox
```

and then from the root of an Astropy repository clone run:

```
$ tox
```

The Astropy tox configuration currently tests against Python versions 2.6, 2.7, 3.2, and 3.3. Tox will automatically skip any Python versions you do not have installed, but best results are achieved if you first install all supported Python versions and make sure they are on your `$PATH`.

Note: Tox creates its virtualenvs in the root of your project under a `.tox` directory (which is automatically ignored by `.gitignore`). It's worth making note of this, however, as it is common practice to sometimes clean up a git repository and delete any untracked files by running the `git clean -dfx` command. As it can take a long time to rebuild the tox virtualenvs you may want to exclude the `.tox` directory from any cleanup. This can be achieved by running `git clean -dfx -e .tox`, though it is probably worth defining a [git alias](#) to do this.

38.2.5 Test-running options

Running parts of the test suite

It is possible to run only the tests for a particular subpackage. For example, to run only the `wcs` tests from the commandline:

```
python setup.py test -P wcs
```

Or from Python:

```
>>> import astropy
>>> astropy.test(package="wcs")
```

You can also specify a single file to test from the commandline:

```
python setup.py test -t astropy/wcs/tests/test_wcs.py
```

When the `-t` option is given a relative path, it is relative to the installed root of astropy. When `-t` is given a relative path to a documentation `.rst` file to test, it is relative to the root of the documentation, i.e. the `docs` directory in the source tree. For example:

```
python setup.py test -t units/index.rst
```

Testing for open files

Astropy can test whether any of the unit tests inadvertently leave any files open. Since this greatly slows down the time it takes to run the tests, it is turned off by default.

To use it from the commandline, do:

```
python setup.py test --open-files
```

To use it from Python, do:

```
>>> import astropy
>>> astropy.test(open_files=True)
```

Test coverage reports

Astropy can use `coverage.py` to generate test coverage reports. To generate a test coverage report, use:

```
python setup.py test --coverage
```

There is a `coveragerc` file that defines files to omit as well as lines to exclude. It is installed along with astropy so that the astropy testing framework can use it. In the source tree, it is at `astropy/tests/coveragerc`.

Running tests in parallel

It is possible to speed up astropy's tests using the `pytest-xdist` plugin. This plugin can be installed using `pip`:

```
pip install pytest-xdist
```

Once installed, tests can be run in parallel using the `'--parallel'` commandline option. For example, to use 4 processes:

```
python setup.py test --parallel=4
```

Pass a negative number to `'--parallel'` to create the same number of processes as cores on your machine.

Similarly, this feature can be invoked from Python:

```
>>> import astropy
>>> astropy.test(parallel=4)
```

38.3 Writing tests

`py.test` has the following test discovery rules:

- `test_*.py` or `*_test.py` files
- Test prefixed classes (without an `__init__` method)
- `test_` prefixed functions and methods

Consult the [test discovery rules](#) for detailed information on how to name files and tests so that they are automatically discovered by `py.test`.

38.3.1 Simple example

The following example shows a simple function and a test to test this function:

```
def func(x):
    """Add one to the argument."""
    return x + 1

def test_answer():
    """Check the return value of func() for an example argument."""
    assert func(3) == 5
```

If we place this in a `test.py` file and then run:

```
py.test test.py
```

The result is:

```
===== test session starts =====
python: platform darwin -- Python 2.7.2 -- pytest-1.1.1
test object 1: /Users/tom/tmp/test.py

test.py F

===== FAILURES =====
_____ test_answer _____

    def test_answer():
>         assert func(3) == 5
E         assert 4 == 5
E         + where 4 = func(3)

test.py:5: AssertionError
===== 1 failed in 0.07 seconds =====
```

38.3.2 Where to put tests

Package-specific tests

Each package should include a suite of unit tests, covering as many of the public methods/functions as possible. These tests should be included inside each sub-package, e.g:

```
astropy/io/fits/tests/
```

tests directories should contain an `__init__.py` file so that the tests can be imported and so that they can use relative imports.

Interoperability tests

Tests involving two or more sub-packages should be included in:

```
astropy/tests/
```

38.3.3 Regression tests

Any time a bug is fixed, and wherever possible, one or more regression tests should be added to ensure that the bug is not introduced in future. Regression tests should include the ticket URL where the bug was reported.

38.3.4 Working with data files

Tests that need to make use of a data file should use the `get_pkg_data_fileobj` or `get_pkg_data_filename` functions. These functions search locally first, and then on the astropy data server or an arbitrary URL, and return a file-like object or a local filename, respectively. They automatically cache the data locally if remote data is obtained, and from then on the local copy will be used transparently.

They also support the use of an MD5 hash to get a specific version of a data file. This hash can be obtained prior to submitting a file to the astropy data server by using the `compute_hash` function on a local copy of the file.

Tests that may retrieve remote data should be marked with the `@remote_data` decorator, or, if a doctest, flagged with the `REMOTE_DATA` flag. Tests marked in this way will be skipped by default by `astropy.test()` to prevent test runs from taking too long. These tests can be run by `astropy.test()` by adding the `remote_data=True` flag. Turn on the remote data tests at the command line with `py.test --remote-data`.

Examples

```
from ...config import get_data_filename
from ...tests.helper import remote_data

def test_1():
    """Test version using a local file."""
    #if filename.fits is a local file in the source distribution
    datafile = get_data_filename('filename.fits')
    # do the test

@remote_data
def test_2():
    """Test version using a remote file."""
    #this is the hash for a particular version of a file stored on the
    #astropy data server.
    datafile = get_data_filename('hash/94935ac31d585f68041c08f87d1a19d4')
    # do the test

def doctest_example():
    """
    >>> datafile = get_data_filename('hash/94935') # doctest: +REMOTE_DATA
    """
    pass
```

The `get_remote_test_data` will place the files in a temporary directory indicated by the `tempfile` module, so that the test files will eventually get removed by the system. In the long term, once test data files become too large, we will need to design a mechanism for removing test data immediately.

38.3.5 Tests that create files

Tests may often be run from directories where users do not have write permissions so tests which create files should always do so in temporary directories. This can be done with the `py.test tmpdir` function argument or with Python's built-in `tempfile` module.

38.3.6 Setting up/Tearing down tests

In some cases, it can be useful to run a series of tests requiring something to be set up first. There are four ways to do this:

Module-level setup/teardown

If the `setup_module` and `teardown_module` functions are specified in a file, they are called before and after all the tests in the file respectively. These functions take one argument, which is the module itself, which makes it very easy to set module-wide variables:

```
def setup_module(module):
    """Initialize the value of NUM."""
    module.NUM = 11

def add_num(x):
    """Add pre-defined NUM to the argument."""
    return x + NUM

def test_42():
    """Ensure that add_num() adds the correct NUM to its argument."""
    added = add_num(42)
    assert added == 53
```

We can use this for example to download a remote test data file and have all the functions in the file access it:

```
import os

def setup_module(module):
    """Store a copy of the remote test file."""
    module.DATAFILE = get_remote_test_data('94935ac31d585f68041c08f87d1a19d4')

def test():
    """Perform test using cached remote input file."""
    f = open(DATAFILE, 'rb')
    # do the test

def teardown_module(module):
    """Clean up remote test file copy."""
    os.remove(DATAFILE)
```

Class-level setup/teardown

Tests can be organized into classes that have their own setup/teardown functions. In the following

```
def add_nums(x, y):
    """Add two numbers."""
    return x + y

class TestAdd42(object):
    """Test for add_nums with y=42."""

    def setup_class(self):
        self.NUM = 42

    def test_1(self):
```

```
        """Test behaviour for a specific input value."""
        added = add_nums(11, self.NUM)
        assert added == 53

    def test_2(self):
        """Test behaviour for another input value."""
        added = add_nums(13, self.NUM)
        assert added == 55

    def teardown_class(self):
        pass
```

In the above example, the `setup_class` method is called first, then all the tests in the class, and finally the `teardown_class` is called.

Method-level setup/teardown

There are cases where one might want setup and teardown methods to be run before and after *each* test. For this, use the `setup_method` and `teardown_method` methods:

```
def add_nums(x, y):
    """Add two numbers."""
    return x + y

class TestAdd42(object):
    """Test for add_nums with y=42."""

    def setup_method(self, method):
        self.NUM = 42

    def test_1(self):
        """Test behaviour for a specific input value."""
        added = add_nums(11, self.NUM)
        assert added == 53

    def test_2(self):
        """Test behaviour for another input value."""
        added = add_nums(13, self.NUM)
        assert added == 55

    def teardown_method(self, method):
        pass
```

Function-level setup/teardown

Finally, one can use `setup_function` and `teardown_function` to define a setup/teardown mechanism to be run before and after each function in a module. These take one argument, which is the function being tested:

```
def setup_function(function):
    pass

def test_1(self):
    """First test."""
    # do test

def test_2(self):
```

```

    """Second test."""
    # do test

def teardown_method(function):
    pass

```

38.3.7 Parametrizing tests

If you want to run a test several times for slightly different values, then it can be advantageous to use the `py.test` option to parametrize tests. For example, instead of writing:

```

def test1():
    assert type('a') == str

def test2():
    assert type('b') == str

def test3():
    assert type('c') == str

```

You can use the `parametrize` decorator to loop over the different inputs:

```

@pytest.mark.parametrize(('letter'), ['a', 'b', 'c'])
def test(letter):
    """Check that the input is a string."""
    assert type(letter) == str

```

38.3.8 Tests requiring optional dependencies

For tests that test functions or methods that require optional dependencies (e.g. Scipy), `pytest` should be instructed to skip the test if the dependencies are not present. The following example shows how this should be done:

```

import pytest

try:
    import scipy
    HAS SCIPY = True
except ImportError:
    HAS SCIPY = False

@pytest.mark.skipif('not HAS SCIPY')
def test_that_uses_scipy():
    ...

```

In this way, the test is run if Scipy is present, and skipped if not. No tests should fail simply because an optional dependency is not present.

38.3.9 Using `py.test` helper functions

If your tests need to use `py.test` helper functions, such as `pytest.raises`, import `pytest` into your test module like so:

```

from ...tests.helper import pytest

```

You may need to adjust the relative import to work for the depth of your module. `tests.helper` imports `pytest` either from the user's system or `extern.pytest` if the user does not have `py.test` installed. This is so that users need not install `py.test` to run AstroPy's tests.

38.3.10 Testing warnings

In order to test that warnings are triggered as expected in certain situations, you can use the `astropy.tests.helper.catch_warnings` context manager. Unlike the `warnings.catch_warnings` context manager in the standard library, this one will reset all warning state before hand so one is assured to get the warnings reported, regardless of what errors may have been emitted by other tests previously. Here is a real-world example:

```
from astropy.tests.helper import catch_warnings

with catch_warnings(MergeConflictWarning) as warning_lines:
    # Test code which triggers a MergeConflictWarning
    out = table.vstack([t1, t2, t4], join_type='outer')

    assert warning_lines[0].category == metadata.MergeConflictWarning
    assert ("In merged column 'a' the 'units' attribute does not match (cm != m)"
           in str(warning_lines[0].message))
```

Note: Within `py.test` there is also the option of using the `recwarn` function argument to test that warnings are triggered. This method has been found to be problematic in at least one case ([pull request 1174](#)) so the `astropy.tests.helper.catch_warnings` context manager is preferred.

38.3.11 Testing with Unicode literals

Python 2 can run code in two modes: by default, string literals are 8-bit `bytes` objects. However, when `from __future__ import unicode_literals` is used, string literals are `unicode` objects. In order to ensure that `astropy` supports user code written in both styles, the testing framework has a special feature to run a module containing tests in both modes. Simply add the comment:

```
# TEST_UNICODE_LITERALS
```

anywhere in the file, and all tests in that file will be tested twice: once in the default mode where string literals are `bytes`, and again where string literals are `unicode`.

38.3.12 Marking blocks of code to exclude from coverage

Blocks of code may be ignored by the coverage testing by adding a comment containing the phrase `pragma: no cover` to the start of the block:

```
if this_rarely_happens: # pragma: no cover
    this_call_is_ignored()
```

Blocks of code that are intended to run only in Python 2.x or 3.x may also be marked so that they will be ignored when appropriate by `coverage.py`:

```
if sys.version_info[0] >= 3: # pragma: py3
    do_it_the_python3_way()
else: # pragma: py2
    do_it_the_python2_way()
```

Using `six.PY3` and `six.PY2` will also automatically exclude blocks from coverage, without requiring the pragma comment:

```
if six.PY3:
    do_it_the_python3_way()
elif six.PY2:
    do_it_the_python2_way()
```

38.4 Writing doctests

A doctest in Python is a special kind of test that is embedded in a function, class, or module's docstring, or in the narrative Sphinx documentation, and is formatted to look like a Python interactive session—that is, they show lines of Python code entered at a `>>>` prompt followed by the output that would be expected (if any) when running that code in an interactive session.

The idea is to write usage examples in docstrings that users can enter verbatim and check their output against the expected output to confirm that they are using the interface properly.

Furthermore, Python includes a `doctest` module that can detect these doctests and execute them as part of a project's automated test suite. This way we can automatically ensure that all doctest-like examples in our docstrings are correct.

The Astropy test suite automatically detects and runs any doctests in the Astropy source code or documentation, or in affiliated packages using the Astropy test running framework. For example doctests and detailed documentation on how to write them, see the full `doctest` documentation.

Note: Since the narrative Sphinx documentation is not installed alongside the astropy source code, it can only be tested by running `python setup.py test`, not by `import astropy; astropy.test()`.

38.4.1 Skipping doctests

Sometimes it is necessary to write examples that look like doctests but that are not actually executable verbatim. An example may depend on some external conditions being fulfilled, for example. In these cases there are a few ways to skip a doctest:

1. Next to the example add a comment like: `# doctest: +SKIP`. For example:

```
>>> import os
>>> os.listdir('.') # doctest: +SKIP
```

In the above example we want to direct the user to run `os.listdir('.')` but we don't want that line to be executed as part of the doctest.

To skip tests that require fetching remote data, use the `REMOTE_DATA` flag instead. This way they can be turned on using the `--remote-data` flag when running the tests:

```
>>> datafile = get_data_filename('hash/94935') # doctest: +REMOTE_DATA
```

2. Astropy's test framework adds support for a special `__doctest_skip__` variable that can be placed at the module level of any module to list functions, classes, and methods in that module whose doctests should not be run. That is, if it doesn't make sense to run a function's example usage as a doctest, the entire function can be skipped in the doctest collection phase.

The value of `__doctest_skip__` should be a list of wildcard patterns for all functions/classes whose doctests should be skipped. For example:

```
__doctest_skip__ = ['myfunction', 'MyClass', 'MyClass.*']
```

skips the doctests in a function called `myfunction`, the doctest for a class called `MyClass`, and all *methods* of `MyClass`.

Module docstrings may contain doctests as well. To skip the module-level doctests include the string `'.'` in `__doctest_skip__`.

To skip all doctests in a module:

```
__doctest_skip__ = ['*']
```

3. In the Sphinx documentation, a doctest section can be skipped by making it part of a `doctest-skip` directive:

```
.. doctest-skip::  
  
    >>> # This is a doctest that will appear in the documentation,  
    >>> # but will not be executed by the testing framework.  
    >>> 1 / 0 # Divide by zero, ouch!
```

It is also possible to skip all doctests below a certain line using a `doctest-skip-all` comment. Note the lack of `::` at the end of the line here:

```
.. doctest-skip-all  
  
All doctests below here are skipped...
```

4. `__doctest_requires__` is a way to list dependencies for specific doctests. It should be a dictionary mapping wildcard patterns (in the same format as `__doctest_skip__`) to a list of one or more modules that should be *importable* in order for the tests to run. For example, if some tests require the `scipy` module to work they will be skipped unless `import scipy` is possible. It is also possible to use a tuple of wildcard patterns as a key in this dict:

```
__doctest_requires__ = {('func1', 'func2'): ['scipy']}
```

Having this module-level variable will require `scipy` to be importable in order to run the doctests for functions `func1` and `func2` in that module.

In the Sphinx documentation, a doctest requirement can be notated with the `doctest-requires` directive:

```
.. doctest-requires:: scipy  
  
    >>> import scipy  
    >>> scipy.hamming(...)
```

38.4.2 Skipping output

One of the important aspects of writing doctests is that the example output can be accurately compared to the actual output produced when running the test.

The doctest system compares the actual output to the example output verbatim by default, but this not always feasible. For example the example output may contain the `__repr__` of an object which displays its id (which will change on each run), or a test that expects an exception may output a traceback.

The simplest way to generalize the example output is to use the ellipses `...`. For example:

```
>>> 1 / 0  
Traceback (most recent call last):  
...
```

```
ZeroDivisionError: integer division or modulo by zero
Traceback (most recent call last):
...
ZeroDivisionError: integer division or modulo by zero
```

This doctest expects an exception with a traceback, but the text of the traceback is skipped in the example output—only the first and last lines of the output are checked. See the `:mod:doctest` documentation for more examples of skipping output.

38.4.3 Handling float output

Some doctests may produce output that contains string representations of floating point values. Floating point representations are often not exact and contain roundoffs in their least significant digits. Depending on the platform the tests are being run on (different Python versions, different OS, etc.) the exact number of digits shown can differ. Because doctests work by comparing strings this can cause such tests to fail.

To address this issue Astropy’s test framework includes support for a `FLOAT_CMP` flag that can be used with doctests. For example:

```
>>> 1.0 / 3.0 # doctest: +FLOAT_CMP
0.333333333333333311
```

When this flag is used, the expected and actual outputs are both parsed to find any floating point values in the strings. Those are then converted to actual Python `float` objects and compared numerically. This means that small differences in representation of roundoff digits will be ignored by the doctest. The values are otherwise compared exactly, so more significant (albeit possibly small) differences will still be caught by these tests.

WRITING COMMAND-LINE SCRIPTS

Command-line scripts in Astropy should follow a consistent scheme to promote readability and compatibility.

The actual script should be in the `/scripts` directory of the Astropy source distribution, and should do nothing aside from importing a `main` function from `astropy` and execute it. This is partly necessary because the “2to3” utility that converts python 2.x code to 3.x does not convert scripts. These scripts should be executable, include `#!/usr/bin/env python` at the top, and should *not* end in `.py`.

The main functions these scripts call should accept an optional single argument that holds the `sys.argv` list, except for the script name (e.g., `argv[1:]`). This function can live in its own module, or be part of a larger module that implements a class or function for `astropy` library use. The `main` function should do very little actual work - it should only parse the arguments and pass those arguments on to some library function so that the library function can be used programmatically when needed. Command-line options can be parsed however desired, but the `argparse` module is recommended when possible, due to its simpler and more flexible interface relative to the older `optparse`. `argparse` is only available in python `>=2.7` and `>=3.2`, however, so it should be imported as `from astropy.util.compat import argparse`.

39.1 Example

Contents of `/scripts/cmdlinescript`

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""An astropy command-line script"""
```

```
import astropy.somepackage.somemod
```

```
astropy.somepackage.somemod.main()
```

Contents of `/astropy/somepackage/somemod.py`

```
def do_something(args, option=False):
    for a in args:
        if option:
            ...do something...
        else:
            ...do something else...

def main(args=None):
    from astropy.utils.compat import argparse

    parser = argparse.ArgumentParser(description='Process some integers.')
    parser.add_argument('-o', '--option', dest='op', action='store_true',
                        help='Some option that turns something on.')
```

```
parser.add_argument('stuff', metavar='S', nargs='+',
                    help='Some input I should be able to get lots of.')

res = parser.parse_args(args)

do_something(res.stuff, res.op)
```

BUILDING ASTROPY AND ITS SUBPACKAGES

The build process currently uses the `setuptools` package to build and install the astropy core (and any affiliated packages that use the template). The user doesn't necessarily need to have `setuptools` installed, as it will automatically bootstrap itself using the `ez_setup.py` file in the source distribution if it isn't installed for the user.

40.1 Astropy-helpers

As of Astropy v0.4, Astropy also uses an external package called `astropy-helpers` to provide some of its build and installation functionality. A copy of `astropy-helpers` is included with the Astropy source distribution, but also includes a mechanism to automatically download bug fixes from PyPI. The reason for providing these helpers as a separate package is that it makes it easier for affiliated packages to take advantage of these same utilities without requiring Astropy to be installed *first*. See [APE4](#) for the full background on this.

`Astropy-helpers` is automatically bootstrapped to the Astropy build/installation script (`setup.py`) via a script called `ah_bootstrap.py` that is imported by `setup.py`. This script will do its best to ensure that the user has an up-to-date copy of `astropy-helpers` before building the package. The auto-upgrade mechanism in particular allows pushing platform-specific fixes for the build process without releasing a new version of Astropy (or any affiliated package that uses `astropy-helpers`).

The behavior of the `ah_bootstrap.py` script can also be modified by options in the project's `setup.cfg` file under a section called `[ah_bootstrap]`. [APE4](#) provides [more details](#).

The `astropy-helpers` distribution provides a Python package called `astropy_helpers`. Code that previously referenced the modules `astropy.setup_helpers` and `astropy.version_helpers` should now depend on `astropy_helpers` and use `astropy_helpers.setup_helpers` and `astropy_helpers.version_helpers` respectively. Likewise, `astropy-helpers` includes tools for building Astropy's documentation. The `astropy.sphinx` package is deprecated in favor of `astropy_helpers.sphinx`. As such, `astropy-helpers` is a dependency of building Astropy's documentation.

40.2 Customizing setup/build for subpackages

As is typical, there is a single `setup.py` file that is used for the whole `astropy` package. To customize setup parameters for a given sub-package, a `setup_package.py` file can be defined inside a package, and if it is present, the setup process will look for the following functions to customize the build process:

- **get_package_data**

This function, if defined, should return a dictionary mapping the name of the subpackage(s) that need package data to a list of data file paths (possibly including wildcards) relative to the path of the package's source code. e.g. if the source distribution has a needed data file `astropy/wcs/tests/data/3d_cd.hdr`, this function should return `{'astropy.wcs.tests': ['data/3d_cd.hdr']}`. See the `package_data` option of the `distutils.core.setup()` function.

It is recommended that all such data be in a directory named `data` inside the package within which it is supposed to be used. This package data should be accessed via the `astropy.utils.data.get_pkg_data_filename` and `astropy.utils.data.get_pkg_data_fileobj` functions.

- **get_extensions**

This provides information for building C or Cython extensions. If defined, it should return a list of `distutils.core.Extension` objects controlling the Cython/C build process (see below for more detail).

- **get_build_options**

This function allows a package to add extra build options. It should return a list of tuples, where each element has:

- *name*: The name of the option as it would appear on the commandline or in the `setup.cfg` file.
- *doc*: A short doc string for the option, displayed by `setup.py build --help`.
- *is_bool* (optional): When `True`, the option is a boolean option and doesn't have an associated value.

Once an option has been added, its value can be looked up using `astropy_helpers.setup_helpers.get_distutils_build_option`.

- **get_external_libraries**

This function declares that the package uses libraries that are included in the astropy distribution that may also be distributed elsewhere on the users system. It should return a list of library names. For each library, a new build option is created, `'--use-system-X'` which allows the user to request to use the system's copy of the library. The package would typically call `astropy_helpers.setup_helpers.use_system_library` from its `get_extensions` function to determine if the package should use the system library or the included one.

- **requires_2to3**

This function declares whether the package requires processing through the `2to3` tool to run on Python 3. If not included, it defaults to `True`. The use of `2to3` is being phased out in astropy, in favor of using `six` instead. See *Writing portable code for Python 2 and 3* for more information.

The `astropy_helpers.setup_helpers` module includes an `update_package_files` function which automatically searches the given source path for `setup_package.py` modules and calls each of the above functions, if they exist. This makes it easy for affiliated packages to use this machinery in their own `setup.py`.

C OR CYTHON EXTENSIONS

Astropy supports using C extensions for wrapping C libraries and Cython for speeding up computationally-intensive calculations. Both Cython and C extension building can be customized using the `get_extensions` function of the `setup_package.py` file. If defined, this function must return a list of `distutils.core.Extension` objects. The creation process is left to the subpackage designer, and can be customized however is relevant for the extensions in the subpackage.

While C extensions must always be defined through the `get_extensions` mechanism, Cython files (ending in `.pyx`) are automatically located and loaded in separate extensions if they are not in `get_extensions`. For Cython extensions located in this way, headers for numpy C functions are included in the build, but no other external headers are included. `.pyx` files present in the extensions returned by `get_extensions` are not included in the list of extensions automatically generated extensions. Note that this allows disabling a Cython file by providing an extension that includes the Cython file, but giving it the special name `'cython_skip'`. Any extension with this package name will not be built by `setup.py`.

Note: If an `Extension` object is provided for Cython source files using the `get_extensions` mechanism, it is very important that the `.pyx` files be given as the `source`, rather than the `.c` files generated by Cython.

41.1 Installing C header files

If your C extension needs to be linked from other third-party C code, you probably want to install its header files alongside the Python module.

1. Create an `include` directory inside of your package for all of the header files.
2. Use the `get_package_data` hook in `setup_package.py` to install those header files. For example, the `astropy.wcs` package has this:

```
def get_package_data():
    return {'astropy.wcs': ['include/*.h']}
```

41.2 Preventing importing at build time

In rare cases, some packages may need to be imported at build time. Unfortunately, anything that requires a C or Cython extension or processing through 2to3 will fail to import until the build phase has completed. In those cases, the `__ASTROPY_SETUP__` variable can be used to determine if the package is being imported as part of the build and choose to not import problematic modules. `__ASTROPY_SETUP__` is inserted into the builtins, and is `True` when inside of astropy's `setup.py` script, and `False` otherwise.

For example, suppose there is a subpackage `foo` that needs to import a module called `version.py` at build time in order to set some version information, and also has a C extension, `process`, that will not be available in the source tree. In this case, `astropy/foo/__init__.py` would probably want to check the value of `_ASTROPY_SETUP_` before importing the C extension:

```
try:
    from . import process
except ImportError:
    if not _ASTROPY_SETUP_:
        raise

from . import version
```

RELEASE PROCEDURES

The current release procedure for Astropy involves a combination of an automated release script and some manual steps. Future versions will automate more of the process, if not all.

42.1 Release Procedure

The automated portion of the Astropy release procedure uses `zest.releaser` to create the tag and update the version. `zest.releaser` is extendable through hook functions—Astropy already includes a couple hook functions to modify the default behavior, but future releases may be further automated through the implementation of additional hook functions. In order to use the hooks, Astropy itself must be *installed* alongside `zest.releaser`. It is recommended to create a `virtualenv` specifically for this purpose.

This may seem like a lot of steps, but most of them won't be necessary to repeat for each release. The advantage of using an automated or semi-automated procedure is that ensures a consistent release process each time.

1. Ensure you have a GPG key pair available for when `git` needs to sign the tag you create for the release. See *Creating a GPG Signing Key and a Signed Tag* for more on this.
2. Update the list of contributors in the `creditsandlicense.rst` file. The easiest way to check this is do:

```
$ git shortlog -s
```

And just add anyone from that list who isn't already credited.

3. Install `virtualenv` if you don't already have it. See the linked `virtualenv` documentation for details. Also, make sure that you have `cython` installed, as you will need it to generate the `.c` files needed for the release.
4. Create and activate a `virtualenv`:

```
$ virtualenv --system-site-packages astropy-release  
$ source astropy-release/bin/activate
```

5. Obtain a *clean* version of the Astropy repository. That is, one where you don't have any intermediate build files. Either use a fresh `git clone` or do `git clean -dfx`.
6. Be sure you're the "master" branch or, for a bug fix release, on the appropriate bug fix branch. For example, if releasing version 0.2.2 make sure to:

```
$ git checkout v0.2.x
```

7. Now install Astropy into the `virtualenv`:

```
$ python setup.py install
```

This is necessary for two reasons. First, the entry points for the releaser scripts need to be available, and these are in the Astropy package. Second, the build process will generate `.c` files from the Cython `.pyx` files, and the `.c` files are necessary for the source distribution.

8. Install `zest.releaser` into the virtualenv; use `--upgrade --force` to ensure that the latest version is installed in the virtualenv (if you're running a `csh` variant make sure to run `rehash` afterwards too):

```
$ pip install zest.releaser --upgrade --force
```

9. Ensure that all changes to the code have been committed, then start the release by running:

```
$ fullrelease
```

10. You will be asked to enter the version to be released. Press enter to accept the default (which will normally be correct) or enter a specific version string. A diff will then be shown of `CHANGES.rst` and `setup.py` showing that a release date has been added to the changelog, and that the version has been updated in `setup.py`. Enter 'Y' when asked to commit these changes.
11. You will then be shown the command that will be run to tag the release. Enter 'Y' to confirm and run the command.
12. When asked "Check out the tag (for tweaks or pypi/distutils server upload)" enter 'N': `zest.releaser` does not offer enough control yet over how the register and upload are performed so we will do this manually until the release scripts have been improved.
13. You will be asked to enter a new development version. Normally the next logical version will be selected—press enter to accept the default, or enter a specific version string. Do not add `".dev"` to the version, as this will be appended automatically (ignore the message that says `".dev0` will be appended"—it will actually be `".dev"` without the 0). For example, if the just-released version was `"0.1"` the default next version will be `"0.2"`. If we want the next version to be, say `"0.1.1"`, or `"1.0"`, then that must be entered manually.
14. You will be shown a diff of `CHANGES.rst` showing that a new section has been added for the new development version, and showing that the version has been updated in `setup.py`. Enter 'Y' to commit these changes.
15. When asked to push the changes to a remote repository, enter 'Y'. This should complete the portion of the process that's automated at this point.
16. Check out the tag of the released version. For example:

```
$ git checkout v0.1
```

17. Create the source distribution by doing:

```
$ python setup.py sdist
```

Copy the produced `.tar.gz` somewhere and verify that you can unpack it, build it, and get all the tests to pass. It would be best to create a new virtualenv in which to do this.

18. Register the release on PyPI with:

```
$ python setup.py register
```

19. Upload the source distribution to PyPI; this is preceded by re-running the `sdist` command, which is necessary for the upload command to know which distribution to upload:

```
$ python setup.py sdist upload
```

20. Go to https://pypi.python.org/pypi?:action=pkg_edit&name=astropy and ensure that only the most recent releases in each actively maintained release line are *not* marked hidden. For example, if v0.3.1 was just released, v0.3 should be hidden. This is so that users only find the latest bugfix releases.

Do not enable “Auto-hide old releases” as that may hide bugfix releases from older release lines that we may still want to make available.

21. Update the “stable” branch to point to the new stable release For example:

```
$ git checkout stable
$ git reset --hard v0.1
$ git push origin stable --force
```

22. Update Readthedocs so that it builds docs for the corresponding github tag. Also verify that the `stable` Readthedocs version builds correctly for the new version (it should trigger automatically once you’ve done the previous step.)
23. If this was a major/minor release (not a bug fix release) create a bug fix branch for this line of release. That is, if the version just released was “v<major>.<minor>.0”, create bug fix branch with the name “v<major>.<minor>.x”. Starting from the commit tagged as the release, just checkout a new branch and push it to the remote server. For example, after releasing version 0.3, do:

```
$ git checkout -b v0.3.x
```

Then edit `setup.py` so that the `VERSION` variable is `'0.3.1.dev'`, and commit that change. Then, do:

```
$ git push upstream v0.3.x
```

Note:

You may need to replace `upstream` here with `astropy` or whatever remote name you use for the main astropy repository.

The purpose of this branch is for creating bug fix releases like “0.3.1” and “0.3.2”, while allowing development of new features to continue in the master branch. Only changesets that fix bugs without making significant API changes should be merged to the bug fix branches.

24. Update [astropy/astropy-website](#) for the new version. Two files need to be updated: `index.rst` has two tags near the top specifying the current release, and the `docs.rst` file should be updated by putting the previous release in as an older version, and updating the “latest developer version” link to point to the new release.
25. Run the `upload_script.py` script in `astropy-website` to update the actual web site.

42.1.1 Modifications for a beta/release candidate release

For major releases with a lot of changes, we sometimes do beta and/or release candidates to have a chance to catch significant bugs before the true release. If the release you are performing is this kind of pre-release, some of the above steps need to be modified. The primary difference is that these releases go on the <http://testpypi.python.org> server instead of the regular PyPI. The testpypi server provides a place to test the release and host it, but never appears anywhere on the regular server. The price is that testpypi is not guaranteed to be up long-term, but for short-term pre-releases, this is no problem.

The primary modifications to the release procedure are:

- When prompted for a version number (step #13), you will need to manually enter something like “1.0b1” or “1.0rc1”. You should follow this numbering scheme (`x.yb#` or `x.y.zrc#`), as it will ensure the release is ordered “before” the main release by various automated tools.
- On steps #18 and #19, where you register and upload to PyPI, it is important that you add the option `-r https://testpypi.python.org/pypi`. This ensures the release information and files are sent to the test server instead of the real PyPI server. This will probably require you to set up a `~/.pypirc` file appropriate for the testpypi server. See <https://wiki.python.org/moin/TestPyPI> for more on how to do this.
- Do not do step #20 or later, as those are tasks for an actual release.

Note: `~/.pypirc` files necessary for uploading to the testpypi server require you to include your password to be able to manage to do `register` properly. This can be insecure, because it means you have to put your PyPI password in a plain-text file. So you’ll want to set the `~/.pypirc` file permissions to be quite restrictive, use a temporary PyPI password just for doing releases, or some other measure to ensure your password remains secure.

42.2 Maintaining Bug Fix Releases

Astropy releases, as recommended for most Python projects, follows a `<major>.<minor>.<micro>` version scheme, where the “micro” version is also known as a “bug fix” release. Bug fix releases should not change any user-visible interfaces. They should only fix bugs on the previous major/minor release and may also refactor internal APIs or include omissions from previous releases—that is, features that were documented to exist but were accidentally left out of the previous release. They may also include changes to docstrings that enhance clarity but do not describe new features (e.g., more examples, typo fixes, etc).

Bug fix releases are typically managed by maintaining one or more bug fix branches separate from the master branch (the release procedure below discusses creating these branches). Typically, whenever an issue is fixed on the Astropy master branch a decision must be made whether this is a fix that should be included in the Astropy bug fix release. Usually the answer to this question is “yes”, though there are some issues that may not apply to the bug fix branch. For example, it is not necessary to backport a fix to a new feature that did not exist when the bug fix branch was first created. New features are never merged into the bug fix branch—only bug fixes; hence the name.

In rare cases a bug fix may be made directly into the bug fix branch without going into the master branch first. This may occur if a fix is made to a feature that has been removed or rewritten in the development version and no longer has the issue being fixed. However, depending on how critical the bug is it may be worth including in a bug fix release, as some users can be slow to upgrade to new major/micro versions due to API changes.

Issues are assigned to an Astropy release by way of the Milestone feature in the GitHub issue tracker. At any given time there are at least two versions under development: The next major/minor version, and the next bug fix release. For example, at the time of writing there are two release milestones open: `v0.2.2` and `v0.3.0`. In this case, `v0.2.2` is the next bug fix release and all issues that should include fixes in that release should be assigned that milestone. Any issues that implement new features would go into the `v0.3.0` milestone—this is any work that goes in the master branch that should not be backported. For a more detailed set of guidelines on using milestones, see [Using Milestones and Labels](#).

42.2.1 Backporting fixes from master

Most fixes are backported using the `git cherry-pick` command, which applies the diff from a single commit like a patch. For the sake of example, say the current bug fix branch is `v0.2.x`, and that a bug was fixed in master in a commit `abcd1234`. In order to backport the fix, simply checkout the `v0.2.x` branch (it’s also good to make sure it’s in sync with the main Astropy repository) and cherry-pick the appropriate commit:

```
$ git checkout v0.2.x
$ git pull upstream v0.2.x
$ git cherry-pick abcd1234
```

Sometimes a cherry-pick does not apply cleanly, since the bug fix branch represents a different line of development. This can be resolved like any other merge conflict: Edit the conflicted files by hand, and then run `git commit` and accept the default commit message. If the fix being cherry-picked has an associated changelog entry in a separate commit make sure to backport that as well.

What if the issue required more than one commit to fix? There are a few possibilities for this. The easiest is if the fix came in the form of a pull request that was merged into the master branch. Whenever GitHub merges a pull request it generates a merge commit in the master branch. This merge commit represents the *full* difference of all the commits in the pull request combined. What this means is that it is only necessary to cherry-pick the merge commit (this requires adding the `-m 1` option to the cherry-pick command). For example, if `5678abcd` is a merge commit:

```
$ git checkout v0.2.x
$ git pull upstream v0.2.x
$ git cherry-pick -m 1 5678abcd
```

In fact, because Astropy emphasizes a pull request-based workflow, this is the *most* common scenario for backporting bug fixes, and the one requiring the least thought. However, if you're not dealing with backporting a fix that was not brought in as a pull request, read on.

See also:

merge-commits-and-cherry-picks for further explanation of the cherry-pick command and how it works with merge commits.

If not cherry-picking a merge commit there are still other options for dealing with multiple commits. The simplest, though potentially tedious, is to simply run the cherry-pick command once for each commit in the correct order. However, as of Git 1.7.2 it is possible to merge a range of commits like so:

```
$ git cherry-pick 1234abcd..56789def
```

This works fine so long as the commits you want to pick are actually congruous with each other. In most cases this will be the case, though some bug fixes will involve followup commits that need to be backported as well. Most bug fixes will have an issue associated with it in the issue tracker, so make sure to reference all commits related to that issue in the commit message. That way it's harder for commits that need to be backported from getting lost.

42.2.2 Making fixes directly to the bug fix branch

As mentioned earlier in this section, in some cases a fix only applies to a bug fix release, and is not applicable in the mainline development. In this case there are two choices:

1. An Astropy developer with commit access to the main Astropy repository may check out the bug fix branch and commit and push your fix directly.
2. **Preferable:** You may also make a pull request through GitHub against the bug fix branch rather than against master. Normally when making a pull request from a branch on your fork to the main Astropy repository GitHub compares your branch to Astropy's master. If you look on the left-hand side of the pull request page, under "base repo: astropy/astropy" there is a drop-down list labeled "base branch: master". You can click on this drop-down and instead select the bug fix branch ("v0.2.x" for example). Then GitHub will instead compare your fix against that branch, and merge into that branch when the PR is accepted.

42.2.3 Preparing the bug fix branch for release

There are two primary steps that need to be taken before creating a bug fix release. The rest of the procedure is the same as any other release as described in *Release Procedure* (although be sure to provide the right version number).

1. Any existing fixes to the issues assigned to the current bug fix release milestone, or labeled with the relevant “backport-x.y.z” label must be merged into the bug fix branch.
2. The Astropy changelog must be updated to list all issues—especially user-visible issues—fixed for the current release. The changelog should be updated in the master branch, and then merged into the bug fix branch. Most issues *should* already have changelog entries for them. But it’s typical to forget this, so if doesn’t exist yet please add one in the process of backporting. See *Updating and Maintaining the Changelog* for more details.

To aid in this process there is a script called `suggest_backports.py` at <https://gist.github.com/embray/4497178>. The script is not perfect and still needs a little work, but it will get most of the work done. For example, if the current bug fix branch is called ‘v0.2.x’ run it like so:

```
$ suggest_backports.py astropy astropy v0.2.x -f backport.sh
```

This will search GitHub for all issues assigned to the next bug fix release milestone that’s associated with the given bug fix branch (‘v0.2.2’ for example), find the commits that fix those issues, and will generate a shell script called `backport.sh` containing all the `git cherry-pick` commands to backport all those fixes.

The `suggest_backports.py` script will typically take a couple minutes to run, but once it’s done simply execute the generated script from within your local clone of the Astropy repository:

```
$ ./backport.sh
```

This will checkout the appropriate bug fix branch (‘v0.2.x’ in this example), do a `git pull upstream v0.2.x` to make sure it’s up to date, and then start doing cherry-picks into the bug fix branch.

Note: As discussed earlier, cherry-pick may result in merge conflicts. If this occurs, the `backport.sh` script will exit and the conflict should be resolved manually, followed by running `git commit`. To resume the `backport.sh` script after the merge conflict, it is currently necessary to edit the script to either remove or comment out the `git cherry-pick` commands that already ran successfully.

The author of the script hopes to improve it in the future to add `git rebase` like functionality, such that running `backport.sh --continue` or `backport.sh --skip` will be possible in such cases.

Warning: It has also been noted that the `suggest_backports.py` script is not perfect, and can either miss issues that need to be backported, and in some cases can report false positives.

It’s always a good idea before finalizing a bug fix release to look on GitHub through the list of closed issues in the release milestone and check that each one has a fix in the bug fix branch. Usually a quick way to do this is for each issue to run:

```
$ git log --oneline <bugfix-branch> | grep #<issue>
```

Most fixes will mention their related issue in the commit message, so this tends to be pretty reliable. Some issues won’t show up in the commit log, however, as their fix is in a separate pull request. Usually GitHub makes this clear by cross-referencing the issue with its PR. A future version of the `suggest_backports.py` script will perform this check automatically.

Finally, not all issues assigned to a release milestone need to be fixed before making that release. Usually, in the interest of getting a release with existing fixes out within some schedule, it’s best to triage issues that won’t be fixed soon to a new release milestone. If the upcoming bug fix release is ‘v0.2.2’, then go ahead and create a ‘v0.2.3’ milestone and reassign to it any issues that you don’t expect to be fixed in time for ‘v0.2.2’.

42.3 Creating a GPG Signing Key and a Signed Tag

One of the main steps in performing a release is to create a tag in the git repository representing the exact state of the repository that represents the version being released. For Astropy we will always use [signed tags](#): A signed tag is annotated with the name and e-mail address of the signer, a date and time, and a checksum of the code in the tag. This information is then signed with a GPG private key and stored in the repository.

Using a signed tag ensures the integrity of the contents of that tag for the future. On a distributed VCS like git, anyone can create a tag of Astropy called “0.1” in their repository—and where it’s easy to monkey around even after the tag has been created. But only one “0.1” will be signed by one of the Astropy project coordinators and will be verifiable with their public key.

42.3.1 Generating a public/private key pair

Git uses GPG to create signed tags, so in order to perform an Astropy release you will need GPG installed and will have to generate a signing key pair. Most *NIX installations come with GPG installed by default (as it is used to verify the integrity of system packages). If you don’t have the `gpg` command, consult the documentation for your system on how to install it.

For OSX, GPG can be installed from MacPorts using `sudo port install gnupg`.

To create a new public/private key pair, simply run:

```
$ gpg --gen-key
```

This will take you through a few interactive steps. For the encryption and expiry settings, it should be safe to use the default settings (I use a key size of 4096 just because what does a couple extra kilobytes hurt?) Enter your full name, preferably including your middle name or middle initial, and an e-mail address that you expect to be active for a decent amount of time. Note that this name and e-mail address must match the info you provide as your git configuration, so you should either choose the same name/e-mail address when you create your key, or update your git configuration to match the key info. Finally, choose a very good pass phrase that won’t be easily subject to brute force attacks.

If you expect to use the same key for some time, it’s good to make a backup of both your public and private key:

```
$ gpg --export --armor > public.key
$ gpg --export-secret-key --armor > private.key
```

Back up these files to a trusted location—preferably a write-once physical medium that can be stored safely somewhere. One may also back up their keys to a trusted online encrypted storage, though some might not find that secure enough—it’s up to you and what you’re comfortable with.

42.3.2 Add your public key to a keyserver

Now that you have a public key, you can publish this anywhere you like—in your e-mail, in a public code repository, etc. You can also upload it to a dedicated public OpenPGP keyserver. This will store the public key indefinitely (until you manually revoke it), and will be automatically synced with other keyservers around the world. That makes it easy to retrieve your public key using the `gpg` command-line tool.

To do this you will need your public key’s keyname. To find this enter:

```
$ gpg --list-keys
```

This will output something like:

```
/path/to/.gnupg/pubring.gpg
-----
pub  4096D/1234ABCD 2012-01-01
uid  4096D/1234ABCD Your Name <your_email>
sub  4096g/567890EF 2012-01-01
```

The 8 digit hex number on the line starting with “pub”—in this example the “1234ABCD” unique keyname for your public key. To push it to a keyserver enter:

```
$ gpg --send-keys 1234ABCD
```

But replace the 1234ABCD with the keyname for your public key. Most systems come configured with a sensible default keyserver, so you shouldn’t have to specify any more than that.

42.3.3 Create a tag

Now test creating a signed tag in git. It’s safe to experiment with this—you can always delete the tag before pushing it to a remote repository:

```
$ git tag -s v0.1 -m "Astropy version 0.1"
```

This will ask for the password to unlock your private key in order to sign the tag with it. Confirm that the default signing key selected by git is the correct one (it will be if you only have one key).

Once the tag has been created, you can verify it with:

```
$ git tag -v v0.1
```

This should output something like:

```
object e8e3e3edc82b02f2088f4e974dbd2fe820c0d934
type commit
tag v0.1
tagger Your Name <your_email> 1339779534 -0400

Astropy version 0.1
gpg: Signature made Fri 15 Jun 2012 12:59:04 PM EDT using DSA key ID 0123ABCD
gpg: Good signature from "Your Name <your_email>"
```

You can use this to verify signed tags from any repository as long as you have the signer’s public key in your keyring. In this case you signed the tag yourself, so you already have your public key.

Note that if you are planning to do a release following the steps below, you will want to delete the tag you just created, because the release script does that for you. You can delete this tag by doing:

```
$ git tag -d v0.1
```

42.4 Creating a MacOS X Installer on a DMG

The `bdist_dmg` command can be used to create a `.dmg` disk image for MacOS X with a `.pkg` installer. In order to do this, you will need to ensure that you have the following dependencies installed:

- Numpy
- Sphinx
- `bdist_mpkg`

To create a `.dmg` file, run:

```
python setup.py bdist_dmg
```

Note that for the actual release version, you should do this with the Python distribution from python.org (not e.g. MacPorts, EPD, etc.). The best way to ensure maximum compatibility is to make sure that Python and Numpy are installed into `/Library/Frameworks/Python.framework` using the latest stable `.dmg` installers available for those packages. In addition, the `.dmg` should be build on a MacOS 10.6 system, to ensure compatibility with 10.6, 10.7, and 10.8.

Before distributing, you should test out an installation of Python, Numpy, and Astropy from scratch using the `.dmg` installers, preferably on a clean virtual machine.

WORKFLOW FOR MAINTAINERS

This page is for maintainers — those of us who merge our own or other peoples' changes into the upstream repository. Being as how you're a maintainer, you are completely on top of the basic stuff in *How to make a code contribution*.

43.1 Integrating changes via the web interface (recommended)

Whenever possible, merge pull requests automatically via the pull request manager on GitHub. Merging should only be done manually if there is a really good reason to do this!

Make sure that pull requests do not contain a messy history with merges, etc. If this is the case, then follow the manual instructions, and make sure the fork is rebased to tidy the history before committing.

43.2 Integrating changes manually

First, check out the `astropy` repository. The instructions in *Tell git where to look for changes in the development version of Astropy* add a remote that has read-only access to the upstream repo. Being a maintainer, you've got read-write access.

It's good to have your upstream remote have a scary name, to remind you that it's a read-write remote:

```
git remote add upstream-rw git@github.com:astropy/astropy.git
git fetch upstream-rw
```

Let's say you have some changes that need to go into trunk (`upstream-rw/master`).

The changes are in some branch that you are currently on. For example, you are looking at someone's changes like this:

```
git remote add someone git://github.com/someone/astropy.git
git fetch someone
git branch cool-feature --track someone/cool-feature
git checkout cool-feature
```

So now you are on the branch with the changes to be incorporated upstream. The rest of this section assumes you are on this branch.

43.2.1 A few commits

If there are only a few commits, consider rebasing to upstream:

```
# Fetch upstream changes
git fetch upstream-rw

# Rebase
git rebase upstream-rw/master
```

Remember that, if you do a rebase, and push that, you'll have to close any github pull requests manually, because github will not be able to detect the changes have already been merged.

43.2.2 A long series of commits

If there are a longer series of related commits, consider a merge instead:

```
git fetch upstream-rw
git merge --no-ff upstream-rw/master
```

The merge will be detected by github, and should close any related pull requests automatically.

Note the `--no-ff` above. This forces git to make a merge commit, rather than doing a fast-forward, so that these set of commits branch off trunk then rejoin the main history with a merge, rather than appearing to have been made directly on top of trunk.

43.2.3 Check the history

Now, in either case, you should check that the history is sensible and you have the right commits:

```
git log --oneline --graph
git log -p upstream-rw/master..
```

The first line above just shows the history in a compact way, with a text representation of the history graph. The second line shows the log of commits excluding those that can be reached from trunk (`upstream-rw/master`), and including those that can be reached from current HEAD (implied with the `..` at the end). So, it shows the commits unique to this branch compared to trunk. The `-p` option shows the diff for these commits in patch form.

43.2.4 Push to trunk

```
git push upstream-rw my-new-feature:master
```

This pushes the `my-new-feature` branch in this repository to the `master` branch in the `upstream-rw` repository.

43.3 Using Milestones and Labels

These guidelines are adapted from [similar guidelines](#) followed by IPython:

- 100% of confirmed issues and new features should have a milestone
- Only the following criteria should result in an issue being closed without a milestone:
 - Not actually an issue (user error, etc.)
 - Duplicate of an existing issue
 - A pull request superceded by a new pull request providing an alternate implementation

- Open issues should only lack a milestone if:
 - More clarification is required
 - Which milestone it belongs in requires some discussion
- Corollary: When an issue is closed without a milestone that means that the issue will not be fixed, or that it was not a real issue at all.
- In general there should be the following open milestones:
 - The next bug fix releases for any still-supported version lines; for example if 0.4 is in development and 0.2.x and 0.3.x are still supported there should be milestones for the next 0.2.x and 0.3.x releases.
 - The next X.Y release, ie. the next minor release; this is generally the next release that all development in master is aimed toward.
 - The next X.Y release +1; for example if 0.3 is the next release, there should also be a milestone for 0.4 for issues that are important, but that we know won't be resolved in the next release.
 - Future—this is for all issues that require attention at some point but for which no immediate solution is in sight.
- Bug fix release milestones should only be used for deferring issues that won't be fixed in the next minor release, or for issues in previous releases that no longer apply to the mainline.
- When in doubt about which milestone to use for an issue, use the next minor release—it can always be moved once it's been more closely reviewed prior to release.
- Active milestones associated with a specific release (eg. v0.3.0) should contain at least one issue with the release label representing the actual task for releasing that version (this also works around the GitHub annoyance that milestones without any open issues are automatically closed).
- Issues that require fixing in the mainline, but that also are confirmed to apply to supported stable version lines should be marked with one or more 'backport-*' labels for each v0.X.Y branch that has the issue.
 - In some cases it may require extra work beyond a simple merge to port bug fixes to older lines of development; if such additional work is required it is not a bad idea to open a “Backport #nnn to v0.X.Y” issue in the appropriate v0.X.Y milestone.

43.4 Updating and Maintaining the Changelog

The Astropy “changelog” is kept in the file `CHANGES.rst` at the root of the repository. As the filename extension suggests this is a reStructured Text file. The purpose of this file is to give a technical, but still user (and developer) oriented overview of what changes were made to Astropy between each public release. The idea is that it's a little more to the point and easier to follow than trying to read through full git log. It lists all new features added between versions, so that a user can easily find out from reading the changelog when a feature was added. Likewise it lists any features or APIs that were changed (and how they were changed) or removed. It also lists all bug fixes. Affiliated packages are encouraged to maintain a similar changelog.

43.4.1 Adding to the changelog

There are two approaches one may take to adding a new entry to the changelog, each with certain pros and cons. Before describing the two specific approaches it should be said that *all* additions to the changelog should be made first in the ‘master’ branch. This is because every release of Astropy includes a copy of the changelog, and it should list all the changes in every prior version of Astropy. For example, when Astropy v0.3.0 is released, in addition to the changes new to that version the changelog should have all the changes from every v0.2.x version (and earlier) released up to that point.

Two approaches for including a changelog entry for a new feature or bug fix are:

- Include the changelog update in the same pull request as the change. That is, assuming this change is being made in a pull request it can include an accurate changelog update along with it.

Pro: An addition to the changelog is just like any other documentation update, and should be part of any atomic change to the software. It can be pulled into master along with the rest of the change.

Con: If many pull requests also include changelog updates, they can quickly conflict with each other and require rebasing. This is not difficult to resolve if the only conflict is in the changelog, but it can still be trouble especially for new contributors.

- Add to the changelog after a change has been merged to master, whether by pull request or otherwise.

Pro: Largely escapes the merge conflict issue.

Cons: Isn't included "atomically" in the merge commit, making it more difficult to keep track of for backporting. Requires new contributors to either make a second pull request or have a developer with push access to the main repository make the commit.

The first approach is probably preferable, especially for core contributors. But the latter approach is acceptable as well.

43.4.2 Changelog format

The exact formatting of the changelog content is a bit loose for now (though it might become stricter if we want to develop more tools around the changelog). The format can be mostly inferred by looking at previous versions. Each release gets its own heading (using the `-` heading marker) with the version and release date. Releases still under development have `(unreleased)` as there is no release date yet.

There are generally up to three subheadings (using the `^` marker): "New Features", "API Changes", "Bug Fixes", and "Other Changes and Additions". The latter is mostly a catch-all for miscellaneous changes, though there's no reason not to make up additional sub-headings if it seems appropriate.

Under each sub-heading, changes are typically grouped according to which sub-package they pertain to. Changes that apply to more than one sub-package or that only apply to support modules like `logging` or `utils` may go under a "Misc" group.

The actual texts of the changelog entries are typically just one to three sentences—they should be easy to glance over. Most entries end with a reference to an issue/pull request number in square brackets.

A single changelog entry may also reference multiple small changes. For example:

```
- Minor documentation fixes and restructuring.  
  [#935, #967, #978, #1004, #1028, #1047]
```

Beyond that, the best advice for updating the changelog is just to look at existing entries for previous releases and copy the format.

HOW TO CREATE AND MAINTAIN AN ASTROPY AFFILIATED PACKAGE

If you run into any problems, don't hesitate to ask for help on the `astropy-dev` mailing list!

44.1 Starting a new package

The `package-template` repository provides a template for packages that are affiliated with the `Astropy` project. This package design mirrors the layout of the main `Astropy` repository, as well as reusing much of the helper code used to organize `Astropy`. The instructions below describe how to take this template and adjust it for your particular affiliated package.

Everywhere below that the text `yourpkg` is shown, replace it with the name of your particular package.

Note: The instructions below assume you are using `git` for version control, as is used by the `Astropy` repository. If this is not the case, hopefully it will be clear from context what to do with your particular VCS.

1. Make sure `Astropy` is installed, as the template depends in part on `Astropy` to do its setup.
2. You may have already done this if you are looking at this file locally, but if not, you will need to obtain a copy of the package template. Assuming you have `git` installed, just do:

```
git clone git://github.com/astropy/package-template.git yourpkg
```

This will download the latest version of the template from `github` and place it in a directory named `yourpkg`.

1. Go into the directory you just created, and open the `setup.cfg` file with your favorite text editor. Edit the settings in the `metadata` section. These values will be used to automatically replace special placeholders in the affiliated package template.
 - (a) Change the `package_name` variable to whatever you decide your package should be named. By tradition/very strong suggestion, python package names should be all lower-case.
 - (b) Change the `description` variable to a short (one or few sentence) description of your package.
 - (c) Add your name and email address by changing the `author` and `author_email` variables.
 - (d) If your affiliated package has a website, change `url` to point to that site. Otherwise, you can leave it pointing to `Astropy` or just delete it.
 - (e) Exit out of your text editor
2. Update the main package docstring in `packagename/__init__.py`.
3. Decide what license you want to use to release your source code. If you don't care and/or are fine with the `Astropy` license, just edit the file `licenses/LICENSE.rst` with your name (or your collaboration's name)

at the top as the licensees. Otherwise, make sure to replace that file with whatever license you prefer, and update the `license` variable in `setup.cfg` to reflect your choice of license. You also may need to update the comment at the top of `packagename/__init__.py` to reflect your choice of license.

4. Take a moment to look over the `packagename/example_mod.py`, `packagename/tests/test_example.py`, `scripts/script_example`, and `packagename/example_c.pyx` files, as well as the `packagename/example_subpkg` directory. These are examples of a pure-python module, a test script, an example command-line script, a [Cython](#) module, and a sub-package, respectively. ([Cython](#) is a way to compile python-like code to C to make it run faster - see the project's web site for details). These are provided as examples of standard way to lay these out. Once you understand these, though, you'll want to delete them (and later replace with your own):

```
git rm scripts/script_example
git rm packagename/example_c.pyx
git rm packagename/tests/test_example.py
git rm -r packagename/example_subpkg
git commit -m "removed examples from package template"
```

5. Optional: If you're hosting your source code on github, you can enable a sphinx extension that will link documentation pages directly to github's web site. To do this, set `edit_on_github` in `setup.cfg` to `True` and set `github_project` to the name of your project on github.
6. Move the main source directory to reflect the name of your package. To tell your DVCS about this move, you should use it, and not `mv` directly, to make the move. For example, with git:

```
git mv packagename yourpkg
```

7. Update the names of the documentation files to match your package's name. First open `docs/index.rst` in a text editor and change the text "`packagename/index.rst`" to e.g., "`yourpkg/index.rst`". Then do:

```
git add docs/index.rst
git mv docs/packagename docs/yourpkg
```

8. Edit this file (`README.rst`) and delete all of this content, and replace it with a short description of your affiliated package.
9. Open `docs/yourpkg/index.rst` and you can start writing the documentation for your package, but at least replace `packagename` in `automodapi::` with your package name.
10. Now tell git to remember the changes you just made:

```
git commit -a -m "Adjusted for new project yourpkg"
```

11. (This step assumes your affiliated package is hosted as part of the astropy organization on Github. If it's instead hosted somewhere else, just adjust the URL in the instructions below to match wherever your repository lives) Now you will want to tell git that it should be pushing and pulling updates to the repository of *your* project, rather than the package template:

```
git remote rename origin template
git remote add upstream git@github.com:astropy/yourpkg.git
```

Now that it is pointing to the correct master, you should push everything up to your project and make sure that your local master is tied to your project rather than the template. You'll only be able to do this if your github repository is empty (if not, add the `-f` option to the `push` command - that will overwrite whatever is there):

```
git push upstream master
git branch master --set-upstream upstream/master
```

12. (optional) If you are adopting the standard workflow used by [Astropy](#) with github, you will also want to set up a fork of the repo on your own account, by going to the Github page <https://github.com/astropy/yourpkg> and clicking the “fork” button on the upper right. Then run the following commands:

```
git remote add origin git@github.com:yourgithubusername/yourpkg.git
git branch master --set-upstream origin/master
```

Now you can push, pull, and branch whatever you want in your local fork without affecting the official version, but when you want to push something up to the main repository, just switch to the appropriate branch and do `git push upstream master`.

Additionally, you can set things up to make it easier to pull future changes to the package template to your affiliated package. Add a remote for the package template:

```
git remote add template git@github.com:astropy/package-template.git
```

Then, each time you want to pull in changes to the package template:

```
git fetch template
git fetch upstream

# Make your master match the upstream master. This will destroy
# any unmerged commits on your master (which you shouldn't be doing
# work on anyway, according to the standard workflow).
git checkout master
git reset --hard upstream/master

# Merge any recent changes from the package-template
git merge template/master

# ...possibly resolve any conflicts...

# Push to upstream master
git push upstream master
```

13. You should register your package on <https://travis-ci.org> and modify the `.travis.yml` file to make the build pass. This will continuously test your package for each commit, even pull requests against your main repository will be automatically tested, so that you notice when something breaks. For further information see [here](#) and for lot's of example `.travis.yml` build configurations see [here](#). Generally you should aim to always have your master git master branch work with the latest stable as well as the latest development version of astropy (i.e. the astropy git master branch) and the same versions of python and numpy supported by astropy. The template `.travis.yml` covers those versions; in some circumstances you may need to limit the versions your package covers.
14. If you register your package with [coveralls.io](#), then you will need to modify the `coveralls --rcfile` line in `.travis.yml` file to replace `packagename` with the name of your package.
15. If you want the documentation for your project to be hosted by [ReadTheDocs](#), then you need to setup an account there. The following entries in “Advanced Settings” for your package on [ReadTheDocs](#) should work:
- activate `Install your project inside a virtualenv using setup.py install`
 - Requirements file: `docs/rtd-pip-requirements`
 - activate `Give the virtual environment access to the global site-packages dir.`

All other settings can stay on their default value.

16. You're now ready to start doing actual work on your affiliated package. You will probably want to read over the developer guidelines of the [Astropy documentation](#), and if you are hosting your code in [GitHub](#), you might

also want to read the [Github help](#) to ensure you know how to push your code to GitHub and some recommended workflows that work for the core Astropy project.

17. Once you have started work on the affiliated package, you should register your package with the Astropy affiliated package registry. Instructions for doing this will be provided on the [Astropy website](#).
18. Good luck with your code and your science!

44.2 Releasing an affiliated package

You can release an affiliated package using the steps given below. In these instructions, we assume that the changelog file is named `CHANGES.rst`, like for the astropy core package. If instead you use Markdown, then you should replace `CHANGES.rst` by `CHANGES.md` in the instructions.

1. Make sure that Travis and any other continuous integration is passing.
2. Update the `CHANGES.rst` file to make sure that all the changes are listed, and update the release date, which should currently be set to `unreleased`, to the current date in `yyyy-mm-dd` format.
3. Update the version number in `setup.py` to the version you're about to release, without the `.dev` suffix (e.g. `v0.1`).
4. Run `git clean -fxd` to remove any untracked files (WARNING: this will permanently remove any files that have not been previously committed, so make sure that you don't need to keep any of these files).
5. Run:

```
python setup.py sdist --format=gztar
```

and make sure that generated file is good to go by going inside `dist`, expanding the tar file, going inside the expanded directory, and running the tests with:

```
python setup.py test
```

You may need to add the `--remote-data` flag or any other flags that you normally add when fully testing your affiliated package.

6. Go back to the root of the directory and remove the generated files with:

```
git clean -fxd
```

7. Add the changes to `CHANGES.rst` and `setup.py`:

```
git add CHANGES.rst setup.py
```

and commit with message:

```
git commit -m "Preparing release <version>"
```

8. Tag commit with `v<version>`, optionally signing with the `-s` option:

```
git tag v<version>
```

9. Change `VERSION` in `setup.py` to next version number, but with a `.dev` suffix at the end (e.g. `v0.2.dev`). Add a new section to `CHANGES.rst` for next version, with a single entry `No changes yet`, e.g.:

```
0.2 (unreleased)
```

```
-----
```

```
- No changes yet
```

10. Add the changes to `CHANGES.rst` and `setup.py`:

```
git add CHANGES.rst setup.py
```

and commit with message:

```
git commit -m "Back to development: <next_version>"
```

11. Check out the release commit with `git checkout v<version>`. Run `git clean -fxd` to remove any non-committed files, then either release with:

```
python setup.py register sdist --format=gztar upload
```

or, if you are concerned about security, you can also use `twine` as described in [these](#) instructions. Either way, check that the entry on PyPI is correct, and that the tarfile is present.

12. Go back to the master branch and push your changes to github:

```
git checkout master
git push --tags origin master
```

Once you have done this, if you use `readthedocs`, trigger a `latest` build then go to the project settings, and under **Versions** you should see the tag you just pushed. Select the tag to activate it, and save.

Note: The instructions above assume that you do not make use of bug fix branches in your workflow. If you do wish to create a bug fix branch, we recommend that you read over the more complete astropy [Release Procedures](#) and adapt these for your package.

FULL CHANGELOG

45.1 0.4.2 (2014-09-23)

45.1.1 Bug Fixes

- `astropy.coordinates`
 - **Angle accepts hours:mins or deg:mins initializers (without seconds).** In these cases float minutes are also accepted.
 - The `repr` for coordinate frames now displays the frame attributes (ex: `ra`, `dec`) in a consistent order. It should be noted that as part of this fix, the `BaseCoordinateFrame.get_frame_attr_names()` method now returns an `OrderedDict` instead of just a `dict`. [#2845]
- `astropy.io.fits`
 - Fixed a crash when reading scaled float data out of a FITS file that was loaded from a string (using `HDUList.fromfile`) rather than from a file. [#2710]
 - Fixed a crash when reading data from an HDU whose header contained an invalid value for the BLANK keyword (eg. a string value instead of an integer as required by the FITS Standard). Invalid BLANK keywords are now warned about, but are otherwise ignored. [#2711]
 - Fixed a crash when reading the header of a tile-compressed HDU if that header contained invalid duplicate keywords resulting in a `KeyError` [#2750]
 - Fixed crash when reading gzip-compressed FITS tables through the `Astropy Table` interface. [#2783]
 - Fixed corruption when writing new FITS files through to gzipped files. [#2794]
 - Fixed crash when writing HDUs made with non-contiguous data arrays to file-like objects. [#2794]
 - It is now possible to create `astropy.io.fits.BinTableHDU` objects with a table with zero rows. [#2916]
- `astropy.io.misc`
 - Fixed a bug that prevented `h5py Dataset` objects from being automatically recognized by `Table.read`. [#2831]
- `astropy.modeling`
 - Make `LevMarLSQFitter` work with `weights` keyword. [#2900]
- `astropy.table`
 - Fixed reference cycle in tables that could prevent `Table` objects from being freed from memory. [#2879]

- Fixed an issue where `Table.pprint()` did not print the header to `stdout` when `stdout` is redirected (say, to a file). [#2878]
- Fixed printing of masked values when a format is specified. [#1026]
- Ensured that numpy ufuncs that return booleans return plain `ndarray` instances, just like the comparison operators. [#2963]
- `astropy.time`
 - Ensure bigendian input to `Time` works on a little-endian machine (and vice versa). [#2942]
- `astropy.units`
 - Ensure unit is kept when adding 0 to quantities. [#2968]
- `astropy.utils`
 - Fixed color printing on Windows with IPython 2.0. [#2878]
- `astropy.vo`
 - Improved error message on Cone Search time out. [#2687]
- `astropy.wcs`

45.1.2 Other Changes and Additions

- Fixed a couple issues with files being inappropriately included and/or excluded from the source archive distributions of Astropy. [#2843, #2854]
- As part of fixing the fact that masked elements of table columns could not be printed when a format was specified, the column format string options were expanded to allow simple specifiers such as `'5.2f'`. [#2898]
- Ensure numpy 1.9 is supported. [#2917]
- Ensure numpy master is supported, by making `np.cbrt` work with quantities. [#2937]

45.2 0.4.1 (2014-08-08)

45.2.1 Bug Fixes

- `astropy.config`
 - Fixed a bug where an unedited configuration file from astropy 0.3.2 would not be correctly identified as unedited. [#2772] This resulted in the warning:

```
WARNING: ConfigurationChangedWarning: The configuration options
in astropy 0.4 may have changed, your configuration file was not
updated in order to preserve local changes. A new configuration
template has been saved to
'~/astropy/config/astropy.0.4.cfg'. [astropy.config.configuration]
```
 - Fixed the error message that is displayed when an old configuration item has moved. Before, the destination section was wrong. [#2772]
 - Added configuration settings for `io.fits`, `io.votable` and `table.jsviewer` that were missing from the configuration file template. [#2772]
 - The configuration template is no longer rewritten on every import of astropy, causing race conditions. [#2805]

- `astropy.convolution`
 - Fixed the multiplication of `Kernel` with numpy floats. [#2174]
- `astropy.coordinates`
 - Distance can now take a list of quantities. [#2261]
 - For in-place operations for `Angle` instances in which the result unit is not an angle, an exception is raised before the instance is corrupted. [#2718]
 - `CartesianPoints` are now deprecated in favor of `CartesianRepresentation`. [#2727]
- `astropy.io.misc`
 - An existing table within an HDF5 file can be overwritten without affecting other datasets in the same HDF5 file by simultaneously using `overwrite=True` and `append=True` arguments to the `Table.write` method. [#2624]
- `astropy.logger`
 - Fixed a crash that could occur in rare cases when (such as in bundled apps) where submodules of the email package are not importable. [#2671]
- `astropy.nddata`
 - `astropy.nddata.NDData()` no longer raises a `ValueError` when passed a numpy masked array which has no masked entries. [#2784]
- `astropy.table`
 - When saving a table to a FITS file containing a unit that is not supported by the FITS standard, a warning rather than an exception is raised. [#2797]
- `astropy.units`
 - By default, `Quantity` and its subclasses will now convert to float also numerical types such as `decimal.Decimal`, which are stored as objects by numpy. [#1419]
 - The units `count`, `pixel`, `voxel` and `dbyte` now output to FITS, OGIP and `VOUnit` formats correctly. [#2798]
- `astropy.utils`
 - Restored missing information from deprecation warning messages from the `deprecated` decorator. [#2811]
 - Fixed support for `staticmethod` deprecation in the `deprecated` decorator. [#2811]
- `astropy.wcs`
 - Fixed a memory leak when `astropy.wcs.WCS` objects are copied [#2754]
 - Fixed a crash when passing `ra_dec_order=True` to any of the `*2world` methods. [#2791]

45.2.2 Other Changes and Additions

- Bundled copy of `astropy-helpers` upgraded to v0.4.1. [#2825]
- General improvements to documentation and docstrings [#2722, #2728, #2742]
- Made it easier for third-party packagers to have Astropy use their own version of the `six` module (so long as it meets the minimum version requirement) and remove the copy bundled with Astropy. See the `astropy/extern/README` file in the source tree. [#2623]

45.3 0.4 (2014-07-16)

45.3.1 New Features

- `astropy.constants`
 - Added `b_wien` to represent Wien wavelength displacement law constant. [#2194]
- `astropy.convolution`
 - Changed the input parameter in `Gaussian1DKernel` and `Gaussian2DKernel` from `width` to `stddev` [#2085].
- `astropy.coordinates`
 - The `coordinates` package has undergone major changes to implement APE5 . These include backwards-incompatible changes, as the underlying framework has changed substantially. See the APE5 text and the package documentation for more details. [#2422]
 - A `position_angle` method has been added to the new `SkyCoord`. [#2487]
 - Updated `Angle.dms` and `Angle.hms` to return `namedtuple`-s instead of regular tuples, and added `Angle.signed_dms` attribute that gives the absolute value of the `d`, `m`, and `s` along with the sign. [#1988]
 - By default, `Distance` objects are now required to be positive. To allow negative values, set `allow_negative=True` in the `Distance` constructor when creating a `Distance` instance.
 - `Longitude` (resp. `Latitude`) objects cannot be used any more to initialize or set `Latitude` (resp. `Longitude`) objects. An explicit conversion to `Angle` is now required. [#2461]
 - The deprecated functions for pre-0.3 coordinate object names like `ICRSCoordinates` have been removed. [#2422]
 - The `rotation_matrix` and `angle_axis` functions in `astropy.coordinates.angles` were made more numerically consistent and are now tested explicitly [#2619]
- `astropy.cosmology`
 - Added `z_at_value` function to find the redshift at which a cosmology function matches a desired value. [#1909]
 - Added `FLRW.differential_comoving_volume` method to give the differential comoving volume at redshift `z`. [#2103]
 - The functional interface is now deprecated in favor of the more-explicit use of methods on cosmology objects. [#2343]
 - Updated documentation to reflect the removal of the functional interface. [#2507]
- `astropy.io.ascii`
 - The `astropy.io.ascii` output formats `latex` and `aastex` accept a dictionary called `latex_dict` to specify options for LaTeX output. It is now possible to specify the table alignment within the text via the `tablealign` keyword. [#1838]
 - If `header_start` is specified in a call to `ascii.get_reader` or any method that calls `get_reader` (e.g. `ascii.read`) but `data_start` is not specified at the same time, then `data_start` is calculated so that the data starts after the header. Before this, the default was that the header line was read again as the first data line [#855 and #1844].
 - A new `csv` format was added as a convenience for handling CSV (comma-separated values) data. [#1935] This format also recognises rows with an inconsistent number of elements. [#1562]

- An option was added to guess the start of data for CDS format files when they do not strictly conform to the format standard. [#2241]
- Added an HTML reader and writer to the `astropy.io.ascii` package. Parsing requires the installation of BeautifulSoup and is therefore an optional feature. [#2160]
- Added support for inputting column descriptions and column units with the `io.ascii.SExtractor` reader. [#2372]
- Allow the use of non-local ReadMe files in the CDS reader. [#2329]
- Provide a mechanism to select how masked values are printed. [#2424]
- Added support for reading multi-aperture daophot file. [#2656]
- `astropy.io.fits`
 - Included a new command-line script called `fitsheader` to display the header(s) of a FITS file from the command line. [#2092]
 - Added new verification options `fix+ignore`, `fix+warn`, `fix+exception`, `silentfix+ignore`, `silentfix+warn`, and `silentfix+exception` which give more control over how to report fixable errors as opposed to unfixable errors.
- `astropy.modeling`
 - Prototype implementation of fitters that treat optimization algorithms separately from fit statistics, allowing new fitters to be created by mixing and matching optimizers and statistic functions. [#1914]
 - Slight overhaul to how inputs to and outputs from models are handled with respect to array-valued parameters and variables, as well as sets of multiple models. See the associated PR and the modeling section of the v0.4 documentation for more details. [#2634]
 - Added a new `SimplexLSQFitter` which uses a downhill simplex optimizer with a least squares statistic. [#1914]
 - Changed `Gaussian2D` model such that `theta` now increases counterclockwise. [#2199]
 - Replaced the `MatrixRotation2D` model with a new model called simply `Rotation2D` which requires only an angle to specify the rotation. The new `Rotation2D` rotates in a counter-clockwise sense whereas the old `MatrixRotation2D` increased the angle clockwise. [#2266, #2269]
 - Added a new `AffineTransformation2D` model which serves as a replacement for the capability of `MatrixRotation2D` to accept an arbitrary matrix, while also adding a translation capability. [#2269]
 - Added `GaussianAbsorption1D` model. [#2215]
 - New `Redshift` model [#2176].
- `astropy.nddata`
 - Allow initialization `NDData` or `StdDevUncertainty` with a `Quantity`. [#2380]
- `astropy.stats`
 - Added flat prior to `binom_conf_interval` and `binned_binom_proportion`
 - Change default in `sigma_clip` from `np.median` to `np.ma.median`. [#2582]
- `astropy.sphinx`
 - Note, the following new features are included in `astropy-helpers` as well:
 - The `automodapi` and `automodsumm` extensions now include sphinx configuration options to write out what `automodapi` and `automodsumm` generate, mainly for debugging purposes. [#1975, #2022]

- Reference documentation now shows functions/class docstrings at the intended user-facing API location rather than the actual file where the implementation is found. [#1826]
- The `automodsumm` extension configuration was changed to generate documentation of class `__call__` member functions. [#1817, #2135]
- `automodapi` and `automodsumm` now have an `:allowed-package-names:` option that make it possible to document functions and classes that are in a different namespace. [#2370]
- `astropy.table`
 - Improved grouped table aggregation by using the `numpy.reduceat()` method when possible. This can speed up the operation by a factor of at least 10 to 100 for large unmasked tables and columns with relatively small group sizes. [#2625]
 - Allow row-oriented data input using a new `rows` keyword argument. [#850]
 - Allow subclassing of `Table` and the component classes `Row`, `Column`, `MaskedColumn`, `TableColumns`, and `TableFormatter`. [#2287]
 - Fix to allow `numpy` integer types as valid indices into tables in Python 3.x [#2477]
 - Remove transition code related to the order change in `Column` and `MaskedColumn` arguments `name` and `data` from Astropy 0.2 to 0.3. [#2511]
 - Change HTML table representation in IPython notebook to show all table columns instead of restricting to 80 column width. [#2651]
- `astropy.time`
 - Mean and apparent sidereal time can now be calculated using the `sidereal_time` method [#1418].
 - The time scale now defaults to UTC if no scale is provided. [#2091]
 - `TimeDelta` objects can have all scales but UTC, as well as, for consistency with time-like quantities, undefined scale (where the scale is taken from the object one adds to or subtracts from). This allows, e.g., to work consistently in TDB. [#1932]
 - Time now supports ISO format strings that end in “Z”. [#2211, #2203]
- `astropy.units`
 - Support for the unit format [Office of Guest Investigator Programs \(OGIP\) FITS files](#) has been added. [#377]
 - The `spectral` equivalency can now handle angular wave number. [#1306 and #1899]
 - Added `one` as a shorthand for `dimensionless_unscaled`. [#1980]
 - Added `dex` and `dB` units. [#1628]
 - Added `temperature()` equivalencies to support conversion between Kelvin, Celsius, and Fahrenheit. [#2209]
 - Added `temperature_energy()` equivalencies to support conversion between electron-volt and Kelvin. [#2637]
 - The runtime of `astropy.units.Unit.compose` is greatly improved (by a factor of 2 in most cases) [#2544]
 - Added `electron` unit. [#2599]
- `astropy.utils`
 - `timer.RunTimePredictor` now uses `astropy.modeling` in its `do_fit()` method. [#1896]
- `astropy.vo`

- A new sub-package, `astropy.vo.samp`, is now available (this was previously the SAMPy package, which has been refactored for use in Astropy). [#1907]
- Enhanced functionalities for `VOSCatalog` and `VOSDatabase`. [#1206]
- `astropy.wcs`
 - `astropy` now requires `wcslib` version 4.23 or later. The version of `wcslib` included with `astropy` has been updated to version 4.23.
 - Bounds checking is now performed on native spherical coordinates. Any out-of-bounds values will be returned as NaN, and marked in the `stat` array, if using the low-level `wcslib` interface such as `astropy.wcs.Wcsprm.p2s`. [#2107]
 - A new method, `astropy.wcs.WCS.compare()`, compares two `wcsprm` structs for equality with varying degrees of strictness. [#2361]
 - New `astropy.wcs.utils` module, with a handful of tools for manipulating WCS objects, including dropping, swapping, and adding axes.
- Misc
 - Includes the new `astropy-helpers` package which separates some of Astropy’s build, installation, and documentation infrastructure out into an independent package, making it easier for Affiliated Packages to depend on these features. `astropy-helpers` replaces/deprecates some of the submodules in the `astropy` package (see API Changes below). See also [APE 4](#) for more details on the motivation behind and implementation of `astropy-helpers`. [#1563]

45.3.2 API Changes

- `astropy.config`
 - The configuration system received a major overhaul, as part of APE3. It is no longer possible to save configuration items from Python, but instead users must edit the configuration file directly. The locations of configuration items have moved, and some have been changed to science state values. The old locations should continue to work until `astropy 0.5`, but deprecation warnings will be displayed. See the [Configuration transition](#) docs for a detailed description of the changes and how to update existing code. [#2094]
- `astropy.io.fits`
 - The `astropy.io.fits.new_table` function is now fully deprecated (though will not be removed for a long time, considering how widely it is used).

Instead please use the more explicit `BinTableHDU.from_columns` to create a new binary table HDU, and the similar `TableHDU.from_columns` to create a new ASCII table. These otherwise accept the same arguments as `new_table` which is now just a wrapper for these.
 - The `.fromstring` classmethod of each HDU type has been simplified such that, true to its namesake, it only initializes an HDU from a string containing its header *and* data.
 - Fixed an issue where header wildcard matching (for example `header['DATE*']`) can be used to match *any* characters that might appear in a keyword. Previously this only matched keywords containing characters in the set `[0-9A-Za-z_]`. Now this can also match a hyphen `-` and any other characters, as some conventions like `HIERARCH` and record-valued keyword cards allow a wider range of valid characters than standard FITS keywords.
 - This will be the *last* release to support the following APIs that have been marked deprecated since Astropy v0.1/PyFITS v3.1:
 - * The `CardList` class, which was part of the old header implementation.

- * The `Card.key` attribute. Use `Card.keyword` instead.
- * The `Card.cardimage` and `Card.ascardimage` attributes. Use simply `Card.image` or `str(card)` instead.
- * The `create_card` factory function. Simply use the normal `Card` constructor instead.
- * The `create_card_from_string` factory function. Use `Card.fromstring` instead.
- * The `upper_key` function. Use `Card.normalize_keyword` method instead (this is not unlikely to be used outside of PyFITS itself, but it was technically public API).
- * The usage of `Header.update` with `Header.update(keyword, value, comment)` arguments. `Header.update` should only be used analogously to `dict.update`. Use `Header.set` instead.
- * The `Header.ascard` attribute. Use `Header.cards` instead for a list of all the `Card` objects in the header.
- * The `Header.rename_key` method. Use `Header.rename_keyword` instead.
- * The `Header.get_history` method. Use `header['HISTORY']` instead (normal keyword lookup).
- * The `Header.get_comment` method. Use `header['COMMENT']` instead.
- * The `Header.toTxtFile` method. Use `header.totextfile` instead.
- * The `Header.fromTxtFile` method. Use `Header.fromtextfile` instead.
- * The `tdump` and `tcreate` functions. Use `tabledump` and `tableload` respectively.
- * The `BinTableHDU.tdump` and `tcreate` methods. Use `BinTableHDU.dump` and `BinTableHDU.load` respectively.
- * The `txtfile` argument to the `Header` constructor. Use `Header.fromfile` instead.
- * The `startColumn` and `endColumn` arguments to the `FITS_record` constructor. These are unlikely to be used by any user code.

These deprecated interfaces will be removed from the development version of Astropy following the v0.4 release (they will still be available in any v0.4.x bugfix releases, however).

- `astropy.modeling`
 - The method computing the derivative of the model with respect to parameters was renamed from `deriv` to `fit_deriv`. [#1739]
 - `ParametricModel` and the associated `Parametric1DModel` and `Parametric2DModel` classes have been renamed `FittableModel`, `Fittable1DModel`, and `Fittable2DModel` respectively. The base `Model` class has subsumed the functionality of the old `ParametricModel` class so that all models support parameter constraints. The only distinction of `FittableModel` is that anything which subclasses it is assumed “safe” to use with Astropy fitters. [#2276]
 - `NonLinearLSQFitter` has been renamed `LevMarLSQFitter` to emphasise that it uses the Levenberg-Marquardt optimization algorithm with a least squares statistic function. [#1914]
 - The `SLSQPFitter` class has been renamed `SLSQPLSQFitter` to emphasize that it uses the Sequential Least Squares Programming optimization algorithm with a least squares statistic function. [#1914]
 - The `Fitter.errorfunc` method has been renamed to the more general `Fitter.objective_function`. [#1914]
- `astropy.nddata`

- Issue warning if unit is changed from a non-trivial value by directly setting `NDData.unit`. [#2411]
- The `mask` and `flag` attributes of `astropy.nddata.NDData` can now be set with any array-like object instead of requiring that they be set with a `numpy.ndarray`. [#2419]
- `astropy.sphinx`
 - Use of the `astropy.sphinx` module is deprecated; all new development of this module is in `astropy_helpers.sphinx` which should be used instead (therefore documentation builds that made use of any of the utilities in `astropy.sphinx` now have `astropy_helpers` as a documentation dependency).
- `astropy.table`
 - The default table printing function now shows a table header row for units if any columns have the unit attribute set. [#1282]
 - Before, an unmasked `Table` was automatically converted to a masked table if generated from a masked `Table` or a `MaskedColumn`. Now, this conversion is only done if explicitly requested or if any of the input values is actually masked. [#1185]
 - The `repr()` function of `astropy.table.Table` now shows the units if any columns have the unit attribute set. [#2180]
 - The semantics of the config options `table.max_lines` and `table.max_width` has changed slightly. If these values are not set in the config file, `astropy` will try to determine the size automatically from the terminal. [#2683]
- `astropy.time`
 - Correct use of UT in TDB calculation [#1938, #1939].
 - `TimeDelta` objects can have scales other than TAI [#1932].
 - Location information should now be passed on via an `EarthLocation` instance or anything that initialises it, e.g., a tuple containing either geocentric or geodetic coordinates. [#1928]
- `astropy.units`
 - `Quantity` now converts input to float by default, as this is physically most sensible for nearly all units [#1776].
 - `Quantity` comparisons with `==` or `!=` now always return `True` or `False`, even if units do not match (for which case a `UnitsError` used to be raised). [#2328]
 - Applying `float` or `int` to a `Quantity` now works for all dimensionless quantities; they are automatically converted to unscaled dimensionless. [#2249]
 - The exception `astropy.units.UnitException`, which was deprecated in `astropy 0.2`, has been removed. Use `astropy.units.UnitError` instead [#2386]
 - Initializing a `Quantity` with a valid number/array with a `unit` attribute now interprets that attribute as the units of the input value. This makes it possible to initialize a `Quantity` from an `Astropy Table` column and have it correctly pick up the units from the column. [#2486]
- `astropy.wcs`
 - `calcFootprint` was deprecated. It is replaced by `calc_footprint`. An optional boolean keyword `center` was added to `calc_footprint`. It controls whether the centers or the corners of the pixels are used in the computation. [#2384]
 - `astropy.wcs.WCS.sip_pix2foc` and `astropy.wcs.WCS.sip_foc2pix` formerly did not conform to the SIP standard: CRPIX was added to the `foc` result so that it could be used as input to

“core FITS WCS”. As of astropy 0.4, CRPIX is no longer added to the result, so the `foo` space is correct as defined in the [SIP convention](#). [#2360]

- `astropy.wcs.UnitConverter`, which was deprecated in astropy 0.2, has been removed. Use the `astropy.units` module instead. [#2386]

- The following methods on `astropy.wcs.WCS`, which were deprecated in astropy 0.1, have been removed [#2386]:

- * `all_pix2sky` -> `all_pix2world`

- * `wcs_pix2sky` -> `wcs_pix2world`

- * `wcs_sky2pix` -> `wcs_world2pix`

- The `naxis1` and `naxis2` attributes and the `get_naxis` method of `astropy.wcs.WCS`, which were deprecated in astropy 0.2, have been removed. Use the shape of the underlying FITS data array instead. [#2386]

- Misc

- The `astropy.setup_helpers` and `astropy.version_helpers` modules are deprecated; any non-critical fixes and development to those modules should be in `astropy_helpers` instead. Packages that use these modules in their `setup.py` should depend on `astropy_helpers` following the same pattern as in the Astropy package template.

45.3.3 Bug Fixes

- `astropy.constants`

- `astropy.constants.Contant` objects can now be deep copied. [#2601]

- `astropy.cosmology`

- The distance modulus function in `astropy.cosmology` can now handle negative distances, which can occur in certain closed cosmologies. [#2008]

- Removed accidental imports of some extraneous variables in `astropy.cosmology` [#2025]

- `astropy.io.ascii`

- `astropy.io.ascii.read` would fail to read lists of strings where some of the strings consisted of just a newline (“`n`”). [#2648]

- `astropy.io.fits`

- Use NaN for missing values in FITS when using `Table.write` for float columns. Earlier the default fill value was close to `1e20`. [#2186]

- Fixes for checksums on 32-bit platforms. Results may be different if writing or checking checksums in “nonstandard” mode. [#2484]

- Additional minor bug fixes ported from PyFITS. [#2575]

- `astropy.io.votable`

- It is now possible to save an `astropy.table.Table` object as a `VOTable` with any of the supported data formats, `tabledata`, `binary` and `binary2`, by using the `tabledata_format` kwarg. [#2138]

- Fixed a crash writing out variable length arrays. [#2577]

- `astropy.nddata`

- Indexing `NDData` in a way that results in a single element returns that element. [#2170]

- Change construction of result of arithmetic and unit conversion to allow subclasses to require the presence of attribute like unit. [#2300]
- Scale uncertainties to correct units in arithmetic operations and unit conversion. [#2393]
- Ensure uncertainty and mask members are copied in arithmetic and `convert_unit_to`. [#2394]
- Mask result of arithmetic if either of the operands is masked. [#2403]
- Copy all attributes of input object if `astropy.nddata.NDData` is initialized with an `NDData` object. [#2406]
- Copy flags to new object in `convert_unit_to`. [#2409]
- Result of `NDData` arithmetic makes a copy of any `WCS` instead of using a reference. [#2410]
- Fix unit handling for multiplication/division and use `astropy.units.Quantity` for units arithmetic. [#2413]
- A masked `NDData` is now converted to a masked array when used in an operation or `ufunc` with a `numpy` array. [#2414]
- An unmasked `NDData` now uses an internal representation of its mask state that `numpy.ma` expects so that an `NDData` behaves as an unmasked array. [#2417]
- `astropy.sphinx`
 - Fix crash in smart resolver when the resolution doesn't work. [#2591]
- `astropy.table`
 - The `astropy.table.Column` object can now use both functions and callable objects as formats. [#2313]
 - Fixed a problem on 64 bit windows that caused errors "expected 'DTYPE_t' but got 'long long'" [#2490]
 - Fix initialisation of `TableColumns` with lists or tuples. [#2647]
 - Fix removal of single column using `remove_columns`. [#2699]
 - Fix a problem that setting a row element within a masked table did not update the corresponding table element. [#2734]
- `astropy.time`
 - Correct UT1->UTC->UT1 round-trip being off by 1 second if UT1 is on a leap second. [#2077]
- `astropy.units`
 - `Quantity.copy` now behaves identically to `ndarray.copy`, and thus supports the `order` argument (for `numpy >= 1.6`). [#2284]
 - Composing base units into identical composite units now works. [#2382]
 - Creating and composing/decomposing units is now substantially faster [#2544]
 - `Quantity` objects now are able to be assigned `NaN` [#2695]
- `astropy.wcs`
 - Astropy now requires `wcslib` version 4.23 or later. The version of `wcslib` included with astropy has been updated to version 4.23.
 - Bug fixes in the projection routines: in `hpxx2s` [the cartesian-to-spherical operation of the HPX projection] relating to bounds checking, bug introduced at `wcslib` 4.20; in `parx2s` and `molx2s` [the cartesian-to-spherical operation of the PAR and MOL projections respectively] relating to setting the stat vector; in

hp \times 2s relating to implementation of the vector API; and in xp \times 2s relating to setting an out-of-bounds value of *phi*.

- In the PCO projection, use alternative projection equations for greater numerical precision near theta == 0. In the COP projection, return an exact result for theta at the poles. Relaxed the tolerance for bounds checking a little in SFL projection.
- Fix a bug allocating insufficient memory in `astropy.wcs.WCS.sub` [#2468]
- A new method, `Wcsprm.bounds_check` (corresponding to `wcslib`'s `wcsbchk`) has been added to control what bounds checking is performed by `wcslib`.
- `WCS.to_header` will now raise a more meaningful exception when the WCS information is invalid or inconsistent in some way. [#1854]
- In `WCS.to_header`, `RESTFRQ` and `RESTWAV` are no longer rewritten if zero. [#2468]
- In `WCS.to_header`, floating point values will now always be written with an exponent or fractional part, i.e. `.0` being appended if necessary to achieve this. [#2468]
- If the C extension for `astropy.wcs` was not built or fails to import for any reason, `import astropy.wcs` will result in an `ImportError`, rather than getting obscure errors once the `astropy.wcs` is used. [#2061]
- When the C extension for `astropy.wcs` is built using a version of `wcslib` already present in the system, the package does not try to install `wcslib` headers under `astropy/wcs/include`. [#2536]
- Fixes an unresolved external symbol error in the `astropy.wcs._wcs` C extension on Microsoft Windows when built with a Microsoft compiler. [#2478]

- Misc

- Running the test suite with `python setup.py test` now works if the path to the source contains spaces. [#2488]
- The version of ERFA included with Astropy is now v1.1.0 [#2497]
- Removed deprecated option from travis configuration and force use of wheels rather than allowing build from source. [#2576]
- The short option `-n` to run tests in parallel was broken (conflicts with the `distutils` built-in option of “dry-run”). Changed to `-j`. [#2566]

45.3.4 Other Changes and Additions

- `python setup.py test --coverage` will now give more accurate results, because the coverage analysis will include early imports of `astropy`. There doesn't seem to be a way to get this to work when doing `import astropy; astropy.test()`, so the `coverage` keyword to `astropy.test` has been removed. Coverage testing now depends only on `coverage.py`, not `pytest-cov`. [#2112]
- The included version of `py.test` has been upgraded to 2.5.1. [#1970]
- The included version of `six.py` has been upgraded to 1.5.2. [#2006]
- Where appropriate, tests are now run both with and without the `unicode_literals` option to ensure that we support both cases. [#1962]
- Running the Astropy test suite from within the IPython REPL is disabled for now due to bad interaction between the test runner and IPython's logging and I/O handler. For now, run the Astropy tests should be run in the basic Python interpreter. [#2684]
- Added support for numerical comparison of floating point values appearing in the output of `doctests` using a `+FLOAT_CMP` `doctest` flag. [#2087]

- A monkey patch is performed to fix a bug in Numpy version 1.7 and earlier where unicode fill values on masked arrays are not supported. This may cause unintended side effects if your application also monkey patches `numpy.ma` or relies on the broken behavior. If unicode support of masked arrays is important to your application, upgrade to Numpy 1.8 or later for best results. [#2059]
- The developer documentation has been extensively rearranged and rewritten. [#1712]
- The `human_time` function in `astropy.utils` now returns strings without zero padding. [#2420]
- The `bdist_dmg` command for `setup.py` has now been removed. [#2553]
- Many broken API links have been fixed in the documentation, and the `nitpick` Sphinx option is now used to avoid broken links in future. [#1221, #2019, #2109, #2161, #2162, #2192, #2200, #2296, #2448, #2456, #2460, #2467, #2476, #2508, #2509]

45.4 0.3.2 (2014-05-13)

45.4.1 Bug Fixes

- `astropy.coordinates`
 - if `sep` argument is specified to be a single character in `sexagisimal_to_string`, it now includes separators only between items [#2183]
 - Ensure comparisons involving `Distance` objects do not raise exceptions; also ensure operations that lead to units other than length return `Quantity`. [#2206, #2250]
 - Multiplication and division of `Angle` objects is now supported. [#2273]
 - Fixed `Angle.to_string` functionality so that negative angles have the correct amount of padding when `pad=True`. [#2337]
 - Mixing strings and quantities in the `Angle` constructor now works. For example: `Angle(['1d', 1. * u.d])`. [#2398]
 - If `Longitude` is given a `Longitude` as input, use its `wrap_angle` by default [#2705]
- `astropy.cosmology`
 - Fixed `format()` compatibility with Python 2.6. [#2129]
 - Be more careful about converting to floating point internally [#1815, #1818]
- `astropy.io.ascii`
 - The CDS reader in `astropy.io.ascii` can now handle multiple description lines in `ReadMe` files. [#2225]
 - When reading a table with values that generate an overflow error during type conversion (e.g. overflowing the native C long type), fall through to using string. Previously this generated an exception [#2234].
 - Some CDS files mark missing values with "----", others with "--". Recognize any string with one to four dashes as null value. [#1335]
- `astropy.io.fits`
 - Allow pickling of `FITS_rec` objects. [#1597]
 - Improved behavior when writing large compressed images on OSX by removing an unnecessary check for platform architecture. [#2345]

- Fixed an issue where Astropy Table objects containing boolean columns were not correctly written out to FITS files. [#1953]
 - Several other bug fixes ported from PyFITS v3.2.3 [#2368]
 - Fixed a crash on Python 2.x when writing a FITS file directly to a StringIO.StringIO object. [#2463]
- `astropy.io.registry`
 - Allow readers/writers with the same name to be attached to different classes. [#2312]
- `astropy.io.votable`
 - By default, floating point values are now written out using `repr` rather than `str` to preserve precision [#2137]
- `astropy.modeling`
 - Fixed the `SIP` and `InverseSIP` models both so that they work in the first place, and so that they return results consistent with the `SIP` functions in `astropy.wcs`. [#2177]
- `astropy.stats`
 - Ensure the `axis` keyword in `astropy.stats.funcs` can now be used for all axes. [#2173]
- `astropy.table`
 - Ensure nameless columns can be printed, using ‘None’ for the header. [#2213]
- `astropy.time`
 - Fixed pickling of `Time` objects. [#2123]
- `astropy.units`
 - `Quantity._repr_latex_()` returns `NotImplementedError` for quantity arrays instead of an uninformative formatting exception. [#2258]
 - Ensure `Quantity.flat` always returns `Quantity`. [#2251]
 - Angstrom unit renders better in MathJax [#2286]
- `astropy.utils`
 - Progress bars will now be displayed inside the IPython qtconsole. [#2230]
 - `data.download_file()` now evaluates `REMOTE_TIMEOUT()` at runtime rather than import time. Previously, setting `REMOTE_TIMEOUT` after import had no effect on the function’s behavior. [#2302]
 - Progressbar will be limited to 100% so that the bar does not exceed the terminal width. The numerical display can still exceed 100%, however.
- `astropy.vo`
 - Fixed `format()` compatibility with Python 2.6. [#2129]
 - Cone Search validation no longer raises `ConeSearchError` for positive RA. [#2240, #2242]
- `astropy.wcs`
 - Fixed a bug where calling `astropy.wcs.Wcsprm.sub` with `WCSSUB_CELESTIAL` may cause memory corruption due to underallocation of a temporary buffer. [#2350]
 - Fixed a memory allocation bug in `astropy.wcs.Wcsprm.sub` and `astropy.wcs.Wcsprm.copy`. [#2439]
- Misc
 - Fixes for compatibility with Python 3.4. [#1945]

- `import astropy; astropy.test()` now correctly uses the same test configuration as `python setup.py test` [#1811]

45.5 0.3.1 (2014-03-04)

45.5.1 Bug Fixes

- `astropy.config`
 - Fixed a bug where `ConfigurationItem.set_temp()` does not reset to default value when exception is raised within `with block`. [#2117]
- `astropy.convolution`
 - Fixed a bug where `_truncation` was left undefined for `CustomKernel`. [#2016]
 - Fixed a bug with `_normalization` when `CustomKernel` input array sums to zero. [#2016]
- `astropy.coordinates`
 - Fixed a bug where using `==` on two array coordinates wouldn't work. [#1832]
 - Fixed bug which caused `len()` not to work for coordinate objects and added a `.shape` property to get appropriately array-like behavior. [#1761, #2014]
 - Fixed a bug where sexagesimal notation would sometimes include exponential notation in the last field. [#1908, #1913]
 - `CompositeStaticMatrixTransform` no longer attempts to reference the undefined variable `self.matrix` during instantiation. [#1944]
 - Fixed pickling of `Longitude`, ensuring `wrap_angle` is preserved [#1961]
 - Allow `sep` argument in `Angle.to_string` to be empty (resulting in no separators) [#1989]
- `astropy.io.ascii`
 - Allow passing unicode delimiters when reading or writing tables. The delimiter must be convertible to pure ASCII. [#1949]
 - Fix a problem when reading a table and renaming the columns to names that already exist. [#1991]
- `astropy.io.fits`
 - Ported all bug fixes from PyFITS 3.2.1. See the PyFITS changelog at <http://pyfits.readthedocs.org/en/v3.2.1/> [#2056]
- `astropy.io.misc`
 - Fixed issues in the HDF5 Table reader/writer functions that occurred on Windows. [#2099]
- `astropy.io.votable`
 - The `write_null_values` kwarg to `VOTable.to_xml`, when set to `False` (the default) would produce non-standard VOTable files. Therefore, this functionality has been replaced by a better understanding that knows which fields in a VOTable may be left empty (only `char`, `float` and `double` in VOTable 1.1 and 1.2, and all fields in VOTable 1.3). The kwarg is still accepted but it will be ignored, and a warning is emitted. [#1809]
 - Printing out a `astropy.io.votable.tree.Table` object using `repr` or `str` now uses the pretty formatting in `astropy.table`, so it's possible to easily preview the contents of a VOTable. [#1766]
- `astropy.modeling`

- Fixed bug in computation of model derivatives in `LinearLSQFitter`. [#1903]
- Raise a `NotImplementedError` when fitting composite models. [#1915]
- Fixed bug in the computation of the `Gaussian2D` model. [#2038]
- Fixed bug in the computation of the `AiryDisk2D` model. [#2093]
- `astropy.sphinx`
 - Added slightly more useful debug info for `AstropyAutosummary`. [#2024]
- `astropy.table`
 - The column string representation for n-dimensional cells with only one element has been fixed. [#1522]
 - Fix a problem that caused `MaskedColumn.__getitem__` to not preserve column metadata. [#1471, #1872]
 - With Numpy prior to version 1.6.2, tables with Unicode columns now sort correctly. [#1867]
 - `astropy.table` can now print out tables with Unicode columns containing non-ascii characters. [#1864]
 - Columns can now be named with Unicode strings, as long as they contain only ascii characters. This makes using `astropy.table` easier on Python 2 when `from __future__ import unicode_literals` is used. [#1864]
 - Allow pickling of `Table`, `Column`, and `MaskedColumn` objects. [#792]
 - Fix a problem where it was not possible to rename columns after sorting or adding a row. [#2039]
- `astropy.time`
 - Fix a problem where scale conversion problem in `TimeFromEpoch` was not showing a useful error [#2046]
 - Fix a problem when converting to one of the formats `unix`, `cxcsec`, `gps` or `plot_date` when the time scale is `UT1`, `TDB` or `TCB` [#1732]
 - Ensure that `delta_ut1_utc` gets calculated when accessed directly, instead of failing and giving a rather obscure error message [#1925]
 - Fix a bug when computing the `TDB` to `TT` offset. The transform routine was using meters instead of kilometers for the Earth vector. [#1929]
 - Increase `__array_priority__` so that `TimeDelta` can convert itself to a `Quantity` also in reverse operations [#1940]
 - Correct hop list from `TCG` to `TDB` to ensure that conversion is possible [#2074]
- `astropy.units`
 - `Quantity` initialisation rewritten for speed [#1775]
 - Fixed minor string formatting issue for dimensionless quantities. [#1772]
 - Fix error for inplace operations on non-contiguous quantities [#1834].
 - The definition of the unit `bar` has been corrected to “1e5 Pascal” from “100 Pascal” [#1910]
 - For units that are close to known units, but not quite, for example due to differences in case, the exception will now include recommendations. [#1870]
 - The generic and FITS unit parsers now accept multiple slashes in the unit string. There are multiple ways to interpret them, but the approach taken here is to convert “m/s/kg” to “m s-1 kg-1”. Multiple slashes are accepted, but discouraged, by the FITS standard, due to the ambiguity of parsing, so a warning is raised when it is encountered. [#1911]

- The use of “angstrom” (with a lower case “a”) is now accepted in FITS unit strings, since it is in common usage. However, since it is not officially part of the FITS standard, a warning will be issued when it is encountered. [#1911]
- Pickling unrecognized units will not raise a `AttributeError`. [#2047]
- `astropy.units` now correctly preserves the precision of fractional powers. [#2070]
- If a `Unit` or `Quantity` is raised to a floating point power that is very close to a rational number with a denominator less than or equal to 10, it is converted to a `Fraction` object to preserve its precision through complex unit conversion operations. [#2070]
- `astropy.utils`
 - Fixed crash in `timer.RunTimePredictor.do_fit`. [#1905]
 - Fixed `astropy.utils.compat.argparse` for Python 3.1. [#2017]
- `astropy.wcs`
 - `astropy.wcs.WCS`, `astropy.wcs.WCS.fix` and `astropy.wcs.find_all_wcs` now have a `translate_units` keyword argument that is passed down to `astropy.wcs.Wcsprm.fix`. This can be used to specify any unsafe translations of units from rarely used ones to more commonly used ones.

Although "S" is commonly used to represent seconds, its translation to "s" is potentially unsafe since the standard recognizes "S" formally as Siemens, however rarely that may be used. The same applies to "H" for hours (Henry), and "D" for days (Debye).

When these sorts of changes are performed, a warning is emitted. [#1854]
 - When a unit is “fixed” by `astropy.wcs.WCS.fix` or `astropy.wcs.Wcsprm.unitfix`, it now correctly reports the `CUNIT` field that was changed. [#1854]
 - `astropy.wcs.Wcs.printwcs` will no longer warn that `cdelt` is being ignored when none was present in the FITS file. [#1845]
 - `astropy.wcs.Wcsprm.set` is called from within the `astropy.wcs.WCS` constructor, therefore any invalid information in the keywords will be raised from the constructor, rather than on a subsequent call to a transformation method. [#1918]
 - Fix a memory corruption bug when using `astropy.wcs.Wcs.sub` with `astropy.wcs.WCSSUB_CELESTIAL`. [#1960]
 - Fixed the `AttributeError` exception that was raised when using `astropy.wcs.WCS.footprint_to_file`. [#1912]
 - Fixed a `NameError` exception that was raised when using `astropy.wcs.validate` or the `wcslint` script. [#2053]
 - Fixed a bug where named WCSes may be erroneously reported as ' ' when using `astropy.wcs.validate` or the `wcslint` script. [#2053]
 - Fixed a bug where error messages about incorrect header keywords may not be propagated correctly, resulting in a “NULL error object in wcslib” message. [#2106]
- Misc
 - There are a number of improvements to make Astropy work better on big endian platforms, such as MIPS, PPC, s390x and SPARC. [#1849]
 - The test suite will now raise exceptions when a deprecated feature of Python or Numpy is used. [#1948]

45.5.2 Other Changes and Additions

- A new function, `astropy.wcs.get_include`, has been added to get the location of the `astropy.wcs` C header files. [#1755]
- The doctests in the `.rst` files in the `docs` folder are now tested along with the other unit tests. This is in addition to the testing of doctests in docstrings that was already being performed. See `docs/development/testguide.rst` for more information. [#1771]
- Fix a problem where import fails on Python 3 if `setup.py` exists in current directory. [#1877]

45.6 0.3 (2013-11-20)

45.6.1 New Features

- General
 - A top-level configuration item, `unicode_output` has been added to control whether the Unicode string representation of certain objects will contain Unicode characters. For example, when `use_unicode` is `False` (default):

```
>>> from astropy import units as u
>>> print(unicode(u.degree))
deg
```

When `use_unicode` is `True`:

```
>>> from astropy import units as u
>>> print(unicode(u.degree))
◦
```

See [handling-unicode](#) for more information. [#1441]
 - * `astropy.utils.misc.find_api_page` is now imported into the top-level. This allows usage like `astropy.find_api_page(astropy.units.Quantity)`. [#1779]
- `astropy.convolution`
 - New class-based system for generating kernels, replacing `make_kernel`. [#1255] The `astropy.nddata.convolution` sub-package has now been moved to `astropy.convolution`. [#1451]
- `astropy.coordinates`
 - Two classes `astropy.coordinates.Longitude` and `astropy.coordinates.Latitude` have been added. These are derived from the new `Angle` class and used for all longitude-like (RA, azimuth, galactic L) and latitude-like coordinates (Dec, elevation, galactic B) respectively. The `Longitude` class provides auto-wrapping capability and `Latitude` performs bounds checking.
 - `astropy.coordinates.Distance` supports conversion to and from distance moduli. [#1472]
 - `astropy.coordinates.SphericalCoordinateBase` and derived classes now support arrays of coordinates, enabling large speed-ups for some operations on multiple coordinates at the same time. These coordinates can also be indexed using standard slicing or any Numpy-compatible indexing. [#1535, #1615]
 - Array coordinates can be matched to other array coordinates, finding the closest matches between the two sets of coordinates (see the `astropy.coordinates.matching.match_coordinates_3d` and `astropy.coordinates.matching.match_coordinates_sky` functions). [#1535]

- `astropy.cosmology`
 - Added support for including massive Neutrinos in the cosmology classes. The Planck (2013) cosmology has been updated to use this. [#1364]
 - Calculations now use and return `Quantity` objects where appropriate. [#1237]
- `astropy.io.ascii`
 - Added support for writing IPAC format tables [#1152].
- `astropy.io.fits`
 - Added initial support for table columns containing pseudo-unsigned integers. This is currently enabled by using the `uint=True` option when opening files; any table columns with the correct `BZERO` value will be interpreted and returned as arrays of unsigned integers. [#906]
 - Upgraded vendored copy of CFITSIO to v3.35, though backwards compatibility back to version v3.28 is maintained.
 - Added support for reading and writing tables using the Q format for columns. The Q format is identical to the P format (variable-length arrays) except that it uses 64-bit integers for the data descriptors, allowing more than 4 GB of variable-length array data in a single table.
 - Some refactoring of the table and `FITS_rec` modules in order to better separate the details of the FITS binary and ASCII table data structures from the HDU data structures that encapsulate them. Most of these changes should not be apparent to users (but see API Changes below).
- `astropy.io.votable`
 - Updated to support the VOTable 1.3 draft. [#433]
 - Added the ability to look up and group elements by their `utype` attribute. [#622]
 - The format of the units of a VOTable file can be specified using the `unit_format` parameter. Note that units are still always written out using the CDS format, to ensure compatibility with the standard.
- `astropy.modeling`
 - Added a new framework for representing and evaluating mathematical models and for fitting data to models. See “What’s New in Astropy 0.3” in the documentation for further details. [#493]
- `astropy.stats`
 - Added robust statistics functions `astropy.stats.funcs.median_absolute_deviation`, `astropy.stats.funcs.biweight_location`, and `astropy.stats.funcs.biweight_midvariance`. [#621]
 - Added `astropy.stats.funcs.signal_to_noise_oir_ccd` for computing the signal to noise ratio for source being observed in the optical/IR using a CCD. [#870]
 - Add `axis=int` option to `astropy.stats.funcs.sigma_clip` to allow clipping along a given axis for multidimensional data. [#1083]
- `astropy.table`
 - New columns can be added to a table via assignment to a non-existing column by name. [#726]
 - Added `join` function to perform a database-like join on two tables. This includes support for inner, left, right, and outer joins as well as metadata merging. [#903]
 - Added `hstack` and `vstack` functions to stack two or more tables. [#937]
 - Tables now have a `.copy` method and include support for `copy` and `deepcopy`. [#1208]

- Added support for selecting and manipulating groups within a table with a database style `group_by` method. [#1424]
 - Table `read` and `write` functions now include rudimentary support reading and writing of FITS tables via the unified reading/writing interface. [#591]
 - The `units` and `dtypes` attributes and keyword arguments in `Column`, `MaskedColumn`, `Row`, and `Table` are now deprecated in favor of the single-tense `unit` and `dtype`. [#1174]
 - Setting a column from a `Quantity` now correctly sets the unit on the `Column` object. [#732]
 - Add `remove_row` and `remove_rows` to remove table rows. [#1230]
 - Added a new `Table.show_in_browser` method that opens a web browser and displays the table rendered as HTML. [#1342]
 - New tables can now be instantiated using a single row from an existing table. [#1417]
- `astropy.time`
 - New `Time` objects can be instantiated from existing `Time` objects (but with different format, scale, etc.) [#889]
 - Added a `Time.now` classmethod that returns the current UTC time, similarly to Python's `datetime.now`. [#1061]
 - Update internal time manipulations so that arithmetic with `Time` and `TimeDelta` objects maintains sub-nanosecond precision over a time span longer than the age of the universe. [#1189]
 - Use `astropy.utils.iers` to provide `delta_ut1_utc`, so that automatic calculation of UT1 becomes possible. [#1145]
 - Add `datetime` format which allows converting to and from standard library `datetime.datetime` objects. [#860]
 - Add `plot_date` format which allows converting to and from the date representation used when plotting dates with `matplotlib` via the `matplotlib.pyplot.plot_date` function. [#860]
 - Add `gps` format (seconds since 1980-01-01 00:00:00 UTC, including leap seconds) [#1164]
 - Add array indexing to `Time` objects [#1132]
 - Allow for arithmetic of multi-element and single-element `Time` and `TimeDelta` objects. [#1081]
 - Allow multiplication and division of `TimeDelta` objects by constants and arrays, as well as changing sign (negation) and taking the absolute value of `TimeDelta` objects. [#1082]
 - Allow comparisons of `Time` and `TimeDelta` objects. [#1171]
 - Support interaction of `Time` and `Quantity` objects that represent a time interval. [#1431]
 - `astropy.units`
 - Added parallax equivalency for length-angle. [#985]
 - Added mass-energy equivalency. [#1333]
 - Added a new-style format method which will use format specifiers (like `0.03f`) in new-style format strings for the `Quantity`'s value. Specifiers which can't be applied to the value will fall back to the entire string representation of the quantity. [#1383]
 - Added support for complex number values in quantities. [#1384]
 - Added new spectroscopic equivalencies for velocity conversions (relativistic, optical, and radio conventions are supported) [#1200]
 - The `spectral` equivalency now also handles wave number.

- The `spectral_density` equivalency now also accepts a Quantity for the frequency or wavelength. It also handles additional flux units.
 - Added Brightness Temperature (antenna gain) equivalency for conversion between T_B and flux density. [#1327]
 - Added percent unit, and allowed any string containing just a number to be interpreted as a scaled dimensionless unit. [#1409]
 - New-style format strings can be used to set the unit output format. For example, `"{0:latex}".format(u.km)` will print with the latex formatter. [#1462]
 - The `Unit.is_equivalent` method can now take a tuple. In this case, the method returns `True` if the unit is equivalent to any of the units listed in the tuple. [#1521]
 - `def_unit` can now take a 2-tuple of names of the form (short, long), where each entry is a list. This allows for handling strange units that might have multiple short names. [#1543]
 - Added `dimensionless_angles` equivalency, which allows conversion of any power of radian to dimensionless. [#1161]
 - Added the ability to enable set of units, or equivalencies that are used by default. Also provided context managers for these cases. [#1268]
 - Imperial units are disabled by default. [#1593, #1662]
 - Added an `astropy.units.add_enabled_units` context manager, which allows creating a temporary context with additional units temporarily enabled in the global units namespace. [#1662]
 - Unit instances now have `.si` and `.cgs` properties a la Quantity. These serve as shortcuts for `Unit.to_system(cgs)[0]` etc. [#1610]
- `astropy.vo`
 - New package added to support Virtual Observatory Simple Cone Search query and service validation. [#552]
 - `astropy.wcs`
 - Fixed attribute error in `astropy.wcs.Wcsprm(lattype->lattyp)` [#1463]
 - Included a new command-line script called `wcslint` and accompanying API for validating the WCS in a given FITS file or header. [#580]
 - Upgraded included version of WCSLIB to 4.19.
 - `astropy.utils`
 - Added a new set of utilities in `astropy.utils.timer` for analyzing the runtime of functions and making runtime predictions for larger inputs. [#743]
 - `ProgressBar` and `Spinner` classes can now be used directly to return generator expressions. [#771]
 - Added `astropy.utils.iers` which allows reading in of IERS A or IERS B bulletins and interpolation in UT1-UTC.
 - Added a function `astropy.utils.find_api_page`—given a class or object from the `astropy` package, this will open that class's API documentation in a web browser. [#663]
 - Data download functions such as `download_file` now accept a `show_progress` argument to suppress console output, and a `timeout` argument. [#865, #1258]
 - `astropy.extern.six`
 - Added `six` for python2/python3 compatibility

- Astropy now uses the ERFA library instead of the IAU SOFA library for fundamental time transformation routines. The ERFA library is derived, with permission, from the IAU SOFA library but is distributed under a BSD license. See `license/ERFA.rst` for details. [#1293]
- `astropy.logger`
 - The Astropy logger now no longer catches exceptions by default, and also only captures warnings emitted by Astropy itself (prior to this change, following an import of Astropy, any warning got re-directed through the Astropy logger). Logging to the Astropy log file has also been disabled by default. However, users of Astropy 0.2 will likely still see the previous behavior with Astropy 0.3 for exceptions and logging to file since the default configuration file installed by 0.2 set the exception logging to be on by default. To get the new behavior, set the `log_exceptions` and `log_to_file` configuration items to `False` in the `astropy.cfg` file. [#1331]

45.6.2 API Changes

- General
 - The configuration option `utils.console.use_unicode` has been moved to the top level and renamed to `unicode_output`. It now not only affects console widgets, such as progress bars, but also controls whether calling `unicode` on certain classes will return a string containing unicode characters.
- `astropy.coordinates`
 - The `astropy.coordinates.Angle` class is now a subclass of `astropy.units.Quantity`. This means it has all of the methods of a `numpy.ndarray`. [#1006]
 - The `astropy.coordinates.Distance` class is now a subclass of `astropy.units.Quantity`. This means it has all of the methods of a `numpy.ndarray`. [#1472]
 - * All angular units are now supported, not just radian, degree and hour, but now arcsecond and arcminute as well. The object will retain its native unit, so when printing out a value initially provided in hours, its `to_string()` will, by default, also be expressed in hours.
 - * The `Angle` class now supports arrays of angles.
 - * To be consistent with `units.Unit`, `Angle.format` has been deprecated and renamed to `Angle.to_string`.
 - * To be consistent with `astropy.units`, all plural forms of unit names have been removed. Therefore, the following properties of `astropy.coordinates.Angle` should be renamed:
 - `radians` -> `radian`
 - `degrees` -> `degree`
 - `hours` -> `hour`
 - * Multiplication and division of two `Angle` objects used to raise `NotImplementedError`. Now they raise `TypeError`.
 - The `astropy.coordinates.Angle` class no longer has a `bounds` attribute so there is no bounds-checking or auto-wrapping at this level. This allows `Angle` objects to be used in arbitrary arithmetic expressions (e.g. coordinate distance computation).
 - The `astropy.coordinates.RA` and `astropy.coordinates.Dec` classes have been removed and replaced with `astropy.coordinates.Longitude` and `astropy.coordinates.Latitude` respectively. These are now used for the components of Galactic and Horizontal (Alt-Az) coordinates as well instead of plain `Angle` objects.

- `astropy.coordinates.angles.rotation_matrix` and `astropy.coordinates.angles.angle_axis` now take a unit kwarg instead of degrees kwarg to specify the units of the angles. `rotation_matrix` will also take the unit from the given `Angle` object if no unit is provided.
 - The `AngularSeparation` class has been removed. The output of the `coordinates.separation()` method is now an `astropy.coordinates.Angle`. [#1007]
 - The coordinate classes have been renamed in a way that remove the `Coordinates` at the end of the class names. E.g., `ICRSCoordinates` from previous versions is now called `ICRS`. [#1614]
 - `HorizontalCoordinates` are now named `AltAz`, to reflect more common terminology.
- `astropy.cosmology`
 - The Planck (2013) cosmology will likely give slightly different (and more accurate) results due to the inclusion of Neutrino masses. [#1364]
 - Cosmology class properties now return `Quantity` objects instead of simple floating-point values. [#1237]
 - The names of cosmology instances are now truly optional, and are set to `None` rather than the name of the class if the user does not provide them. [#1705]
 - `astropy.io.ascii`
 - In the `read` method of `astropy.io.ascii`, empty column values in an ASCII table are now treated as missing values instead of the previous treatment as a zero-length string `""`. This now corresponds to the behavior of other table readers like `numpy.genfromtxt`. To restore the previous behavior set `fill_values=None` in the call to `ascii.read()`. [#919]
 - The `read` and `write` methods of `astropy.io.ascii` now have a `format` argument for specifying the file format. This is the preferred way to choose the format instead of the `Reader` and `Writer` arguments. [#961]
 - The `include_names` and `exclude_names` arguments were removed from the `BaseHeader` initializer, and now instead handled by the reader and writer classes directly. [#1350]
 - Allow numeric and otherwise unusual column names when reading a table where the `format` argument is specified, but other format details such as the delimiter or quote character are being guessed. [#1692]
 - When reading an ASCII table using the `Table.read()` method, the default has changed from `guess=False` to `guess=True` to allow auto-detection of file format. This matches the default behavior of `ascii.read()`.
 - `astropy.io.fits`
 - The `astropy.io.fits.new_table` function is marked “pending deprecation”. This does not mean it will be removed outright or that its functionality has changed. It will likely be replaced in the future for a function with similar, if not subtly different functionality. A better, if not slightly more verbose approach is to use `pyfits.FITS_rec.from_columns` to create a new `FITS_rec` table—this has the same interface as `pyfits.new_table`. The difference is that it returns a plain `FITS_rec` array, and not an HDU instance. This `FITS_rec` object can then be used as the `data` argument in the constructors for `BinTableHDU` (for binary tables) or `TableHDU` (for ASCII tables). This is analogous to creating an `ImageHDU` by passing in an image array. `pyfits.FITS_rec.from_columns` is just a simpler way of creating a FITS-compatible recarray from a FITS column specification.
 - The `updateHeader`, `updateHeaderData`, and `updateCompressedData` methods of the `CompDataHDU` class are pending deprecation and moved to internal methods. The operation of these methods depended too much on internal state to be used safely by users; instead they are invoked automatically in the appropriate places when reading/writing compressed image HDUs.

- The `CompDataHDU.compData` attribute is pending deprecation in favor of the clearer and more PEP-8 compatible `CompDataHDU.compressed_data`.
 - The constructor for `CompDataHDU` has been changed to accept new keyword arguments. The new keyword arguments are essentially the same, but are in underscore_separated format rather than camelCase format. The old arguments are still pending deprecation.
 - The internal attributes of HDU classes `_hdrLoc`, `_datLoc`, and `_datSpan` have been replaced with `_header_offset`, `_data_offset`, and `_data_size` respectively. The old attribute names are still pending deprecation. This should only be of interest to advanced users who have created their own HDU subclasses.
 - The following previously deprecated functions and methods have been removed entirely: `createCard`, `createCardFromString`, `upperKey`, `ColDefs.data`, `setExtensionNameCaseSensitive`, `_File.getFile`, `_TableBaseHDU.get_coldefs`, `Header.has_key`, `Header.ascardlist`.
 - Interfaces that were pending deprecation are now fully deprecated. These include: `create_card`, `create_card_from_string`, `upper_key`, `Header.get_history`, and `Header.get_comment`.
 - The `.name` attribute on HDUs is now directly tied to the HDU's header, so that if `.header['EXTNAME']` changes so does `.name` and vice-versa.
- `astropy.io.registry`
 - Identifier functions for reading/writing `Table` and `NDData` objects should now accept `(origin, *args, **kwargs)` instead of `(origin, args, kwargs)`. [#591]
 - Added a new `astropy.io.registry.get_formats` function for listing registered I/O formats and details about their readers/writers. [#1669]
 - `astropy.io.votable`
 - Added a new option `use_names_over_ids` option to use when converting from `VOTable` objects to `Astropy Tables`. This can prevent a situation where column names are not preserved when converting from a `VOTable`. [#609]
 - `astropy.nddata`
 - The `astropy.nddata.convolution` sub-package has now been moved to `astropy.convolution`, and the `make_kernel` function has been removed. (the kernel classes should be used instead) [#1451]
 - `astropy.stats.funcs`
 - For `sigma_clip`, the `maout` optional parameter has been removed, and the function now always returns a masked array. A new boolean parameter `copy` can be used to indicate whether the input data should be copied (`copy=True`, default) or used by reference (`copy=False`) in the output masked array. [#1083]
 - `astropy.table`
 - The first argument to the `Column` and `MaskedColumn` classes is now the data array—the name argument has been changed to an optional keyword argument. [#840]
 - Added support for instantiating a `Table` from a list of dict, each one representing a single row with the keys mapping to column names. [#901]
 - The plural 'units' and 'dtypes' have been switched to 'unit' and 'dtype' where appropriate. The original attributes are still present in this version as deprecated attributes, but will be removed in the next version. [#1174]

- The `copy` methods of `Column` and `MaskedColumn` were changed so that the first argument is now `order='C'`. This is required for compatibility with Numpy 1.8 which is currently in development. [#1250]
- Comparing a column (with `==` or `!=`) to a scalar, an array, or another column now always returns a boolean Numpy array (which is a masked array if either of the arguments in the comparison was masked). This is in contrast to the previous behavior, which in some cases returned a boolean Numpy array, and in some cases returned a boolean `Column` object. [#1446]
- `astropy.time`
 - For consistency with `Quantity`, the attributes `val` and `is_scalar` have been renamed to `value` and `isscalar`, respectively, and the attribute `vals` has been dropped. [#767]
 - The double-float64 internal representation of time is used more efficiently to enable better accuracy. [#366]
 - Format and scale arguments are now allowed to be case-insensitive. [#1128]
- `astropy.units`
 - The `Quantity` class now inherits from the Numpy array class, and includes the following API changes [#929]:
 - * Using `float(...)`, `int(...)`, and `long(...)` on a quantity will now only work if the quantity is dimensionless and unscaled.
 - * All Numpy ufuncs should now treat units correctly (or raise an exception if not supported), rather than extract the value of quantities and operate on this, emitting a warning about the implicit loss of units.
 - * When using relevant Numpy ufuncs on dimensionless quantities (e.g. `np.exp(h * nu / (k_B * T))`), or combining dimensionless quantities with Python scalars or plain Numpy arrays `1 + v / c`, the dimensionless `Quantity` will automatically be converted to an unscaled dimensionless `Quantity`.
 - * When initializing a quantity from a value with no unit, it is now set to be dimensionless and unscaled by default. When initializing a `Quantity` from another `Quantity` and with no unit specified in the initializer, the unit is now taken from the unit of the `Quantity` being initialized from.
 - Strings are no longer allowed as the values for `Quantities`. [#1005]
 - `Quantities` are always comparable with zero regardless of their units. [#1254]
 - The exception `astropy.units.UnitsException` has been renamed to `astropy.units.UnitsError` to be more consistent with the naming of built-in Python exceptions. [#1406]
 - Multiplication with and division by a string now always returns a `Unit` (rather than a `Quantity` when the string was first) [#1408]
 - Imperial units are disabled by default.
- `astropy.wcs`
 - For those including the `astropy.wcs` C headers in their project, they should now include it as:

```
#include "astropy_wcs/astropy_wcs_api.h"
```

instead of:

```
#include "astropy_wcs_api.h"
```

[#1631]
- The `--enable-legacy` option for `setup.py` has been removed. [#1493]

45.6.3 Bug Fixes

- `astropy.io.ascii`
 - The `write()` function was ignoring the `fill_values` argument. [#910]
 - Fixed an issue in `DefaultSplitter.join` where the `delimiter` attribute was ignored when writing the CSV. [#1020]
 - Fixed writing of IPAC tables containing null values. [#1366]
 - When a table with no header row was read without specifying the format and using the `names` argument, then the first row could be dropped. [#1692]
- `astropy.io.fits`
 - Binary tables containing compressed images may, optionally, contain other columns unrelated to the tile compression convention. Although this is an uncommon use case, it is permitted by the standard.
 - Reworked some of the file I/O routines to allow simpler, more consistent mapping between OS-level file modes ('rb', 'wb', 'ab', etc.) and the more "PyFITS-specific" modes used by PyFITS like "readonly" and "update". That is, if reading a FITS file from an open file object, it doesn't matter as much what "mode" it was opened in so long as it has the right capabilities (read/write/etc.) Also works around bugs in the Python io module in 2.6+ with regard to file modes.
 - Fixed a long-standing issue where writing binary tables did not correctly write the TFORMn keywords for variable-length array columns (they omitted the max array length parameter of the format). This was thought fixed in an earlier version, but it was only fixed for compressed image HDUs and not for binary tables in general.
- `astropy.nddata`
 - Fixed crash when trying to multiple or divide `NDData` objects with uncertainties. [#1547]
- `astropy.table`
 - Using a list of strings to index a table now correctly returns a new table with the columns named in the list. [#1454]
 - Inequality operators now work properly with `Column` objects. [#1685]
- `astropy.time`
 - Time scale and format attributes are now shown when calling `dir()` on a `Time` object. [#1130]
- `astropy.wcs`
 - Fixed assignment to string-like WCS attributes on Python 3. [#956]
- `astropy.units`
 - Fixed a bug that caused the order of multiplication/division of plain Numpy arrays with `Quantities` to matter (i.e. if the plain array comes first the units were not preserved in the output). [#899]
 - Directly instantiated `CompositeUnits` were made printable without crashing. [#1576]
- Misc
 - Fixed various modules that hard-coded `sys.stdout` as default arguments to functions at import time, rather than using the runtime value of `sys.stdout`. [#1648]
 - Minor documentation fixes and enhancements [#922, #1034, #1210, #1217, #1491, #1492, #1498, #1582, #1608, #1621, #1646, #1670, #1756]
 - Fixed a crash that could sometimes occur when running the test suite on systems with platform names containing non-ASCII characters. [#1698]

45.6.4 Other Changes and Additions

- General
 - Astropy now follows the PSF Code of Conduct. [#1216]
 - Astropy’s test suite now tests all doctests in inline docstrings. Support for running doctests in the reST documentation is planned to follow in v0.3.1.
 - Astropy’s test suite can be run on multiple CPUs in parallel, often greatly improving runtime, using the `--parallel` option. [#1040]
 - A warning is now issued when using Astropy with Numpy < 1.5—much of Astropy may still work in this case but it shouldn’t be expected to either. [#1479]
 - Added automatic download/build/installation of Numpy during Astropy installation if not already found. [#1483]
 - Handling of metadata for the `NDData` and `Table` classes has been unified by way of a common `MetaData` descriptor—it allows instantiating an object with metadata of any mapping type, and subsequently prevents replacing the mapping stored in the `.meta` attribute (only direct updates to that object are allowed). [#1686]
- `astropy.coordinates`
 - Angles containing out of bounds minutes or seconds (eg. 60) can be parsed—the value modulo 60 is used with carry to the hours/minutes, and a warning is issued rather than raising an exception. [#990]
- `astropy.io.fits`
 - The new compression code also adds support for the `ZQUANTIZ` and `ZDITHER0` keywords added in more recent versions of this FITS Tile Compression spec. This includes support for lossless compression with GZIP. (#198) By default no dithering is used, but the `SUBTRACTIVE_DITHER_1` and `SUBTRACTIVE_DITHER_2` methods can be enabled by passing the correct constants to the `quantize_method` argument to the `CompImageHDU` constructor. A seed can be manually specified, or automatically generated using either the system clock or checksum-based methods via the `dither_seed` argument. See the documentation for `CompImageHDU` for more details.
 - Images compressed with the Tile Compression standard can now be larger than 4 GB through support of the Q format.
 - All HDUs now have a `.ver .level` attribute that returns the value of the `EXTVAL` and `EXTLEVEL` keywords from that HDU’s header, if the exist. This was added for consistency with the `.name` attribute which returns the `EXTNAME` value from the header.
 - Then `Column` and `ColDefs` classes have new `.dtype` attributes which give the Numpy dtype for the column data in the first case, and the full Numpy compound dtype for each table row in the latter case.
 - There was an issue where new tables created defaulted the values in all string columns to ‘0.0’. Now string columns are filled with empty strings by default—this seems a less surprising default, but it may cause differences with tables created with older versions of PyFITS or Astropy.
- `astropy.io.misc`
 - The HDF5 reader can now refer to groups in the path as well as datasets; if given a group, the first dataset in that group is read. [#1159]
- `astropy.nddata`
 - `NDData` objects have more helpful, though still rudimentary `__str__` and `__repr__` displays. [#1313]
- `astropy.units`

- Added ‘cycle’ unit. [#1160]
- Extended units supported by the CDS formatter/parser. [#1468]
- Added unicode and LaTeX symbols for liter. [#1618]
- `astropy.wcs`
 - Redundant SCAMP distortion parameters are removed with SIP distortions are also present. [#1278]
 - Added iterative implementation of `all_world2pix` that can be reliably inverted. [#1281]

45.7 0.2.5 (2013-10-25)

45.7.1 Bug Fixes

- `astropy.coordinates`
 - Fixed incorrect string formatting of Angles using `precision=0`. [#1319]
 - Fixed string formatting of Angles using `decimal=True` which ignored the `precision` argument. [#1323]
 - Fixed parsing of format strings using appropriate unicode characters instead of the ASCII – for minus signs. [#1429]
- `astropy.io.ascii`
 - Fixed a crash in the IPAC table reader when the `include/exclude_names` option is set. [#1348]
 - Fixed writing AAS_{Te}x tables to honor the `tabletype` option. [#1372]
- `astropy.io.fits`
 - Improved round-tripping and preservation of manually assigned column attributes (`TNULLn`, `TSCALn`, etc.) in table HDU headers. (Note: This issue was previously reported as fixed in Astropy v0.2.2 by mistake; it is not fixed until v0.3.) [#996]
 - Fixed a bug that could cause a segfault when trying to decompress an compressed HDU whose contents are truncated (due to a corrupt file, for example). This still causes a Python traceback but better than a segfault. [#1332]
 - Newly created `CompImageHDU` HDUs use the correct value of the `DEFAULT_COMPRESSION_TYPE` module-level constant instead of hard-coding “RICE_1” in the header.
 - Fixed a corner case where when extra memory is allocated to compress an image, it could lead to unnecessary in-memory copying of the compressed image data and a possible memory leak through Numpy.
 - Fixed a bug where assigning from an mmap’_d array in one FITS file over the old (also mmap’_d) array in another FITS file failed to update the destination file. Corresponds to PyFITS issue 25.
 - Some miscellaneous documentation fixes.
- `astropy.io.votable`
 - Added a warning for when a VOTable 1.2 file contains no `RESOURCES` elements (at least one should be present). [#1337]
 - Fixed a test failure specific to MIPS architecture caused by an errant floating point warning. [#1179]
- `astropy.nddata.convolution`
 - Prevented in-place modification of the input arrays to `convolve()`. [#1153]

- `astropy.table`
 - Added HTML escaping for string values in tables when outputting the table as HTML. [#1347]
 - Added a workaround in a bug in Numpy that could cause a crash when accessing a table row in a masked table containing `dtype=object` columns. [#1229]
 - Fixed an issue similar to the one in #1229, but specific to unmasked tables. [#1403]
- `astropy.units`
 - Improved error handling for unparseable units and fixed parsing CDS units without mantissas in the exponent. [#1288]
 - Added a physical type for spectral flux density. [#1410]
 - Normalized conversions that should result in a scale of exactly 1.0 to round off slight floating point imprecisions. [#1407]
 - Added support in the CDS unit parser/formatter for unusual unit prefixes that are nonetheless required to be supported by that convention. [#1426]
 - Fixed the parsing of `sqrt()` in unit format strings which was returning `unit ** 2` instead of `unit ** 0.5`. [#1458]
- `astropy.wcs`
 - When passing a single array to the wcs transformation functions, (`astropy.wcs.Wcs.all_pix2world`, etc.), its second dimension must now exactly match the number of dimensions in the transformation. [#1395]
 - Improved error message when incorrect arguments are passed to `WCS.wcs_world2pix`. [#1394]
 - Fixed a crash when trying to read WCS from FITS headers on Python 3.3 in Windows. [#1363]
 - Only headers that are required as part of the WCSLIB C API are installed by the package, per request of system packagers. [#1666]
- Misc
 - Fixed crash when the `COLUMNS` environment variable is set to a non-integer value. [#1291]
 - Fixed a bug in `ProgressBar.map` where `multiprocess=True` could cause it to hang on waiting for the process pool to be destroyed. [#1381]
 - Fixed a crash on Python 3.2 when affiliated packages try to use the `astropy.utils.data.get_pkg_data_*` functions. [#1256]
 - Fixed a minor path normalization issue that could occur on Windows in `astropy.utils.data.get_pkg_data_filename`. [#1444]
 - Fixed an annoyance where configuration items intended only for testing showed up in users' `astropy.cfg` files. [#1477]
 - Prevented crashes in exception logging in unusual cases where no traceback is associated with the exception. [#1518]
 - Fixed a crash when running the tests in unusual environments where `sys.stdout.encoding` is `None`. [#1530]
 - Miscellaneous documentation fixes and improvements [#1308, #1317, #1377, #1393, #1362, #1516]

45.7.2 Other Changes and Additions

- Astropy installation now requests `setuptools >= 0.7` during build/installation if neither `distribute` or `setuptools >= 0.7` is already installed. In other words, if `import setuptools` fails, `ez_setup.py` is used to bootstrap the latest `setuptools` (rather than using `distribute_setup.py` to bootstrap the now obsolete `distribute` package). [#1197]
- When importing Astropy from a source checkout without having built the extension modules first an `ImportError` is raised rather than a `SystemExit` exception. [#1269]

45.8 0.2.4 (2013-07-24)

45.8.1 Bug Fixes

- `astropy.coordinates`
 - Fixed the angle parser to support parsing the string “1 degree”. [#1168]
- `astropy.cosmology`
 - Fixed a crash in the `comoving_volume` method on non-flat cosmologies when passing it an array of redshifts.
- `astropy.io.ascii`
 - Fixed a bug that prevented saving changes to the comment symbol when writing changes to a table. [#1167]
- `astropy.io.fits`
 - Added a workaround for a bug in 64-bit OSX that could cause truncation when writing files greater than 2^{32} bytes in size. [#839]
- `astropy.io.votable`
 - Fixed incorrect reading of tables containing multiple `<RESOURCE>` elements. [#1223]
- `astropy.table`
 - Fixed a bug where `Table.remove_column` and `Table.rename_column` could cause a masked table to lose its masking. [#1120]
 - Fixed bugs where subclasses of `Table` did not preserve their class in certain operations. [#1142]
 - Fixed a bug where slicing a masked table did not preserve the mask. [#1187]
- `astropy.units`
 - Fixed a bug where the `.si` and `.cgs` properties of dimensionless `Quantity` objects raised a `ZeroDivisionError`. [#1150]
 - Fixed a bug where multiple subsequent calls to the `.decompose()` method on array quantities applied a scale factor each time. [#1163]
- Misc
 - Fixed an installation crash that could occur sometimes on Debian/Ubuntu and other *NIX systems where `pkg_resources` can be installed without installing `setuptools`. [#1150]
 - Updated the `distribute_setup.py` bootstrapper to use `setuptools >= 0.7` when installing on systems that don't already have an up to date version of `distribute/setuptools`. [#1180]

- Changed the `version.py` template so that Astropy affiliated packages can (and they should) use their own `cython_version.py` and `utils._compiler` modules where appropriate. This issue only pertains to affiliated package maintainers. [#1198]
- Fixed a corner case where the default config file generation could crash if building with matplotlib but *not* Sphinx installed in a virtualenv. [#1225]
- Fixed a crash that could occur in the logging module on systems that don't have a default preferred encoding (in particular this happened in some versions of PyCharm). [#1244]
- The Astropy log now supports passing non-string objects (and calling `str()` on them by default) to the logging methods, in line with Python's standard logging API. [#1267]
- Minor documentation fixes [#582, #696, #1154, #1194, #1212, #1213, #1246, #1252]

45.8.2 Other Changes and Additions

- `astropy.cosmology`
 - Added a new `Plank13` object representing the Plank 2013 results. [#895]
- `astropy.units`
 - Performance improvements in initialization of `Quantity` objects with a large number of elements. [#1231]

45.9 0.2.3 (2013-05-30)

45.9.1 Bug Fixes

- `astropy.time`
 - Fixed inaccurate handling of leap seconds when converting from UTC to UNIX timestamps. [#1118]
 - Tightened required accuracy in many of the time conversion tests. [#1121]
- Misc
 - Fixed a regression that was introduced in v0.2.2 by the fix to issue #992 that was preventing installation of Astropy affiliated packages that use Astropy's setup framework. [#1124]

45.10 0.2.2 (2013-05-21)

45.10.1 Bug Fixes

- `astropy.io`
 - Fixed issues in both the `fits` and `votable` sub-packages where array byte order was not being handled consistently, leading to possible crashes especially on big-endian systems. [#1003]
- `astropy.io.fits`
 - When an error occurs opening a file in `fitsdiff` the exception message will now at least mention which file had the error.
 - Fixed a couple cases where creating a new table using `TDIMn` in some of the columns could cause a crash.

- Slightly refactored how tables containing variable-length array columns are handled to add two improvements: Fixes an issue where accessing the data after a call to the `astropy.io.fits.getdata` convenience function caused an exception, and allows the VLA data to be read from an existing mmap of the FITS file.
- Fixed a bug on Python 3 where attempting to open a non-existent file on Python 3 caused a seemingly unrelated traceback.
- Fixed an issue in the tests that caused some tests to fail if Astropy is installed with read-only permissions.
- Fixed a bug where instantiating a `BinTableHDU` from a numpy array containing boolean fields converted all the values to `False`.
- Fixed an issue where passing an array of integers into the constructor of `Column()` when the column type is floats of the same byte width caused the column array to become garbled.
- Fixed inconsistent behavior in creating CONTINUE cards from byte strings versus unicode strings in Python 2—CONTINUE cards can now be created properly from unicode strings (so long as they are convertible to ASCII).
- Fixed a bug in parsing HIERARCH keywords that do not have a space after the first equals sign (before the value).
- Prevented extra leading whitespace on HIERARCH keywords from being treated as part of the keyword.
- Fixed a bug where HIERARCH keywords containing lower-case letters was mistakenly marked as invalid during header validation along with an ancillary issue where the `Header.index()` method did not work correctly with HIERARCH keywords containing lower-case letters.
- Disallowed assigning NaN and Inf floating point values as header values, since the FITS standard does not define a way to represent them in. Because this is undefined, the previous behavior did not make sense and produced invalid FITS files. [#954]
- Fixed an obscure issue that can occur on systems that don't have flush to memory-mapped files implemented (namely GNU Hurd). [#968]
- `astropy.io.votable`
 - Stopped deprecation warnings from the `astropy.io.votable` package that could occur during setup. [#970]
 - Fixed an issue where INFO elements were being incorrectly dropped when occurring inside a TABLE element. [#1000]
 - Fixed obscure test failures on MIPS platforms. [#1010]
- `astropy.nddata.convolution`
 - Fixed an issue in `make_kernel()` when using an Airy function kernel. Also removed the superfluous 'brickwall' option. [#939]
- `astropy.table`
 - Fixed a crash that could occur when adding a row to an empty (rowless) table with masked columns. [#973]
 - Made it possible to assign to one table row from the value of another row, effectively making it easier to copy rows, for example. [#1019]
- `astropy.time`
 - Added appropriate `__copy__` and `__deepcopy__` behavior; this omission caused a seemingly unrelated error in FK5 coordinate separation. [#891]
- `astropy.units`

- Fixed an issue where the `isiterable()` utility returned `True` for quantities with scalar values. Added an `__iter__` method for the `Quantity` class and fixed `isiterable()` to catch false positives. [#878]
 - Fixed previously undefined behavior when multiplying a unit by a string. [#949]
 - Added ‘time’ as a physical type—this was a simple omission. [#959]
 - Fixed issues with pickling unit objects so as to play nicer with the multiprocessing module. [#974]
 - Made it more difficult to accidentally override existing units with a new unit of the same name. [#1070]
 - Added several more physical types and units that were previously omitted, including ‘mass density’, ‘specific volume’, ‘molar volume’, ‘momentum’, ‘angular momentum’, ‘angular speed’, ‘angular acceleration’, ‘electric current’, ‘electric current density’, ‘electric field strength’, ‘electric flux density’, ‘electric charge density’, ‘permittivity’, ‘electromagnetic field strength’, ‘radiant intensity’, ‘data quantity’, ‘bandwidth’; and ‘knots’, ‘nautical miles’, ‘becquerels’, and ‘curies’ respectively. [#1072]
- Misc
 - Fixed a permission error that could occur when running `astropy.test()` on Python 3 when Astropy is installed as root. [#811]
 - Made it easier to filter warnings from the `convolve()` function and from `Quantity` objects. [#853]
 - Fixed a crash that could occur in Python 3 when generation of the default config file fails during setup. [#952]
 - Fixed an unrelated error message that could occur when trying to import `astropy` from a source checkout without having build the extension modules first. This issue was claimed to be fixed in v0.2.1, but the fix itself had a bug. [#971]
 - Fixed a crash that could occur when running the `build_sphinx` setup command in Python 3. [#977]
 - Added a more helpful error message when trying to run the `setup.py build_sphinx` command when Sphinx is not installed. [#1027]
 - Minor documentation fixes and restructuring. [#935, #967, #978, #1004, #1028, #1047]

45.10.2 Other Changes and Additions

- Some performance improvements to the `astropy.units` package, in particular improving the time it takes to import the sub-package. [#1015]

45.11 0.2.1 (2013-04-03)

45.11.1 Bug Fixes

- `astropy.coordinates`
 - Fixed encoding errors that could occur when formatting coordinate objects in code using `from __future__ import unicode_literals`. [#817]
 - Fixed a bug where the minus sign was dropped when string formatting dms coordinates with -0 degrees. [#875]
- `astropy.io.fits`
 - Properly supports the `ZQUANTIZ` keyword used to support quantization level—this includes working support for lossless GZIP compression of images.

- Fixed support for opening gzipped FITS files in a writeable mode. [#256]
- Added a more helpful exception message when trying to read invalid values from a table when the required `TNULLn` keyword is missing. [#309]
- More refactoring of the tile compression handling to work around a potential memory access violation that was particularly prevalent on Windows. [#507]
- Fixed an integer size mismatch in the compression module that could affect 32-bit systems. [#786]
- Fixed malformatting of the `TFORMn` keywords when writing compressed image tables (they omitted the max array length parameter from the variable-length array format).
- Fixed a crash that could occur when writing a table containing multi- dimensional array columns from an existing file into a new file.
- Fixed a bug in `fitsdiff` that reported two header keywords containing NaN as having different values.
- `astropy.io.votable`
 - Fixed links to the `astropy.io.votable` documentation in the VOTable validator output. [#806]
 - When reading VOTables containing integers that are out of range for their column type, display a warning rather than raising an exception. [#825]
 - Changed the default string format for floating point values for better round-tripping. [#856]
 - Fixed opening VOTables through the `Table.read()` interface for tables that have no names. [#927]
 - Fixed creation of VOTables from an Astropy table that does not have a data mask. [#928]
 - Minor documentation fixes. [#932]
- `astropy.nddata.convolution`
 - Added better handling of `inf` values to the `convolve_fft` family of functions. [#893]
- `astropy.table`
 - Fixed silent failure to assign values to a row on multiple columns. [#764]
 - Fixed various buggy behavior when viewing a table after sorting by one of its columns. [#829]
 - Fixed using `numpy.where()` with table indexing. [#838]
 - Fixed a bug where opening a remote table with `Table.read()` could cause the entire table to be downloaded twice. [#845]
 - Fixed a bug where `MaskedColumn` no longer worked if the column being masked is renamed. [#916]
- `astropy.units`
 - Added missing capability for array `Quantity`s to be initializable by a list of `Quantity`s. [#835]
 - Fixed the definition of year and lightyear to be in terms of Julian year per the IAU definition. [#861]
 - “degree” was removed from the list of SI base units. [#863]
- `astropy.wcs`
 - Fixed `TypeError` when calling `WCS.to_header_string()`. [#822]
 - Added new method `WCS.all_world2pix` for converting from world coordinates to pixel space, including inversion of the astrometric distortion correction. [#1066, #1281]
- Misc
 - Fixed a minor issue when installing with `./setup.py develop` on a fresh git clone. This is likely only of interest to developers on Astropy. [#725]

- Fixes a crash with `ImportError: No module named 'astropy.version'` when running `setup.py` from a source checkout for the first time on OSX with Python 3.3. [#820]
- Fixed an installation issue where running `./setup.py install` or when installing with pip the `.astropy` directory gets created in the home directory of the user running the command. The user's `.astropy` directory should only be created when they use Astropy, not when they install it. [#867]
- Fixed an exception when creating a `ProgressBar` with a “total” of 0. [#752]
- Added better documentation of behavior that can occur when trying to import the astropy package from within a source checkout without first building the extension modules. [#795, #864]
- Added link to the installation instructions in the README. [#797]
- Catches segfaults in `xmllint` which can occur sometimes and is otherwise out of our control. [#803]
- Minor changes to the documentation template. [#805]
- Fixed a minor exception handling bug in `download_file()`. [#808]
- Added cleanup of any temporary files if an error occurs in `download_file()`. [#857]
- Filesystem free space is checked for before attempting to download a file with `download_file()`. [#858]
- Fixed package data locating to work across symlinks—required to work with some OS packaging layouts. [#827]
- Fixed a bug when building Cython extensions where hidden files containing `.pyx` extensions could cause the build to crash. This can be an issue with software and filesystems that autogenerate hidden files. [#834]
- Fixed bug that could cause a “script” called `README.rst` to be installed in a bin directory. [#852]
- Fixed some miscellaneous and mostly rare reference leaks caught by `cpychecker`. [#914]

45.11.2 Other Changes and Additions

- Added logo and branding for Windows binary installers. [#741]
- Upgraded included version `libexpat` to 2.1.0. [#781]
- ~25% performance improvement in unit composition/decomposition. [#836]
- Added previously missing LaTeX formatting for `L_sun` and `R_sun`. [#841]
- `ConfigurationItems` now have a more useful and informative `__repr__` and improved documentation for how to use them. [#855]
- Added a friendlier error message when trying to import astropy from a source checkout without first building the extension modules inplace. [#864]
- `py.test` now outputs more system information for help in debugging issues from users. [#869]
- Added unit definitions “mas” and “uas” for “milliarcsecond” and “microarcsecond” respectively. [#892]

45.12 0.2 (2013-02-19)

45.12.1 New Features

This is a brief overview of the new features included in Astropy 0.2—please see the “What’s New” section of the documentation for more details.

- `astropy.coordinates`
 - This new subpackage contains a representation of celestial coordinates, and provides a wide range of related functionality. While fully-functional, it is a work in progress and parts of the API may change in subsequent releases.
- `astropy.cosmology`
 - Update to include cosmologies with variable dark energy equations of state. (This introduces some API incompatibilities with the older Cosmology objects).
 - Added parameters for relativistic species (photons, neutrinos) to the `astropy.cosmology` classes. The current treatment assumes that neutrinos are massless. [#365]
 - Add a WMAP9 object using the final (9-year) WMAP parameters from Hinshaw et al. 2013. It has also been made the default cosmology. [#629, #724]
- `astropy.table` I/O infrastructure for custom readers/writers implemented. [#305]
 - Added support for reading/writing HDF5 files [#461]
 - Added support for masked tables with missing or invalid data [#451]
- New `astropy.time` sub-package. [#332]
- New `astropy.units` sub-package that includes a class for units (`astropy.units.Unit`) and scalar quantities that have units (`astropy.units.Quantity`). [#370, #445]

This has the following effects on other sub-packages:

 - In `astropy.wcs`, the `wcs.cunit` list now takes and returns `astropy.units.Unit` objects. [#379]
 - In `astropy.nddata`, units are now stored as `astropy.units.Unit` objects. [#382]
 - In `astropy.table`, units on columns are now stored as `astropy.units.Unit` objects. [#380]
 - In `astropy.constants`, constants are now stored as `astropy.units.Quantity` objects. [#529]
- `astropy.io.ascii`
 - Improved integration with the `astropy.table` `Table` class so that table and column metadata (e.g. keywords, units, description, formatting) are directly available in the output table object. The CDS, DAOPhot, and IPAC format readers now provide this type of integrated metadata.
 - Changed to using `astropy.table` masked tables instead of NumPy masked arrays for tables with missing values.
 - Added SExtractor table reader to `astropy.io.ascii` [#420]
 - Removed the Memory reader class which was used to convert data input passed to the `write` function into an internal table. Instead `write` instantiates an `astropy` `Table` object using the data input to `write`.
 - Removed the `NumpyOutputter` as the output of reading a table is now always a `Table` object.
 - Removed the option of supplying a function as a column output formatter.
 - Added a new `strip_whitespace` keyword argument to the `write` function. This controls whether whitespace is stripped from the left and right sides of table elements before writing. Default is `True`.
 - Fixed a bug in reading IPAC tables with null values.
- Generalized I/O infrastructure so that `astropy.nddata` can also have custom readers/writers [#659]
- `astropy.wcs`
 - From updating the the underlying `wcslib` 4.16:

- * When `astropy.wcs.WCS` constructs a default coordinate representation it will give it the special name “DEFAULTS”, and will not report “Found one coordinate representation”.

45.12.2 Other Changes and Additions

- A configuration file with all options set to their defaults is now generated when astropy is installed. This file will be pulled in as the users’ astropy configuration file the first time they `import astropy`. [#498]
- Astropy doc themes moved into `astropy.sphinx` to allow affiliated packages to access them.
- Added expanded documentation for the `astropy.cosmology` sub-package. [#272]
- Added option to disable building of “legacy” packages (pyfits, vo, etc.).
- The value of the astronomical unit (au) has been updated to that adopted by IAU 2012 Resolution B2, and the values of the pc and kpc constants have been updated to reflect this. [#368]
- Added links to the documentation pages to directly edit the documentation on GitHub. [#347]
- Several updates merged from `pywcs` into `astropy.wcs` [#384]:
 - Improved the reading of distortion images.
 - Added a new option to choose whether or not to write SIP coefficients.
 - Uses the `relax` option by default so that non-standard keywords are allowed. [#585]
- Added HTML representation of tables in IPython notebook [#409]
- Rewrote CFITSIO-based backend for handling tile compression of FITS files. It now uses a standard CFITSIO instead of heavily modified pieces of CFITSIO as before. Astropy ships with its own copy of CFITSIO v3.30, but system packagers may choose instead to strip this out in favor of a system-installed version of CFITSIO. This corresponds to PyFITS ticket 169. [#318]
- Moved `astropy.config.data` to `astropy.utils.data` and re-factored the I/O routines to separate out the generic I/O code that can be used to open any file or resource from the code used to access Astropy-related data. The ‘core’ I/O routine is now `get_readable_fileobj`, which can be used to access any local as well as remote data, supports caching, and can decompress gzip and bzip2 files on-the-fly. [#425]
- Added a classmethod to `astropy.coordinates.coordsystems.SphericalCoordinatesBase` that performs a name resolve query using Sesame to retrieve coordinates for the requested object. This works for any subclass of `SphericalCoordinatesBase`, but requires an internet connection. [#556]
- `astropy.nddata.convolution` removed requirement of PyFFTW3; uses Numpy’s FFT by default instead with the added ability to specify an FFT implementation to use. [#660]

45.12.3 Bug Fixes

- `astropy.io.ascii`
 - Fixed crash when pprinting a row with INDEF values. [#511]
 - Fixed failure when reading DAOphot files with empty keyword values. [#666]
- `astropy.io.fits`
 - Improved handling of scaled images and pseudo-unsigned integer images in compressed image HDUs. They now work more transparently like normal image HDUs with support for the `do_not_scale_image_data` and `uint` options, as well as `scale_back` and `save_backup`. The `.scale()` method works better too. Corresponds to PyFITS ticket 88.

- Permits non-string values for the EXTNAME keyword when reading in a file, rather than throwing an exception due to the malformatting. Added verification for the format of the EXTNAME keyword when writing. Corresponds to PyFITS ticket 96.
- Added support for EXTNAME and EXTVER in PRIMARY HDUs. That is, if EXTNAME is specified in the header, it will also be reflected in the `.name` attribute and in `fits.info()`. These keywords used to be verboten in PRIMARY HDUs, but the latest version of the FITS standard allows them. Corresponds to PyFITS ticket 151.
- HCOMPRESS can again be used to compress data cubes (and higher-dimensional arrays) so long as the tile size is effectively 2-dimensional. In fact, compatible tile sizes will automatically be used even if they're not explicitly specified. Corresponds to PyFITS ticket 171.
- Fixed a bug that could cause a deadlock in the filesystem on OSX when reading the data from certain types of FITS files. This only occurred when used in conjunction with Numpy 1.7. [#369]
- Added support for the optional `endcard` parameter in the `Header.fromtextfile()` and `Header.totextfile()` methods. Although `endcard=False` was a reasonable default assumption, there are still text dumps of FITS headers that include the END card, so this should have been more flexible. Corresponds to PyFITS ticket 176.
- Fixed a crash when running `fitsdiff` on two empty (that is, zero row) tables. Corresponds to PyFITS ticket 178.
- Fixed an issue where opening a FITS file containing a random group HDU in update mode could result in an unnecessary rewriting of the file even if no changes were made. This corresponds to PyFITS ticket 179.
- Fixed a crash when generating diff reports from diffs using the `ignore_comments` options. Corresponds to PyFITS ticket 181.
- Fixed some bugs with WCS Paper IV record-valued keyword cards:
 - * Cards that looked kind of like RVKCs but were not intended to be were over-permissively treated as such—commentary keywords like COMMENT and HISTORY were particularly affected. Corresponds to PyFITS ticket 183.
 - * Looking up a card in a header by its standard FITS keyword only should always return the raw value of that card. That way cards containing values that happen to valid RVKCs but were not intended to be will still be treated like normal cards. Corresponds to PyFITS ticket 184.
 - * Looking up a RVKC in a header with only part of the field-specifier (for example “DP1.AXIS” instead of “DP1.AXIS.1”) was implicitly treated as a wildcard lookup. Corresponds to PyFITS ticket 184.
- Fixed a crash when diffing two FITS files where at least one contains a compressed image HDU which was not recognized as an image instead of a table. Corresponds to PyFITS ticket 187.
- Fixed a bug where opening a file containing compressed image HDUs in ‘update’ mode and then immediately closing it without making any changes caused the file to be rewritten unnecessarily.
- Fixed two memory leaks that could occur when writing compressed image data, or in some cases when opening files containing compressed image HDUs in ‘update’ mode.
- Fixed a bug where `ImageHDU.scale(option='old')` wasn't working at all—it was not restoring the image to its original BSCALE and BZERO values.
- Fixed a bug when writing out files containing zero-width table columns, where the TFIELDS keyword would be updated incorrectly, leaving the table largely unreadable.
- Fixed a minor string formatting issue.
- Fixed bugs in the backwards compatibility layer for the `CardList.index` and `CardList.count` methods. Corresponds to PyFITS ticket 190.

- Improved `__repr__` and text file representation of cards with long values that are split into CONTINUE cards. Corresponds to PyFITS ticket 193.
- Fixed a crash when trying to assign a long (> 72 character) value to blank (‘’) keywords. This also changed how blank keywords are represented—there are still exactly 8 spaces before any commentary content can begin; this *may* affect the exact display of header cards that assumed there could be fewer spaces in a blank keyword card before the content begins. However, the current approach is more in line with the requirements of the FITS standard. Corresponds to PyFITS ticket 194.
- `astropy.io.votable`
 - The `Table` class now maintains a single array object which is a Numpy masked array. For variable-length columns, the object that is stored there is also a Numpy masked array.
 - Changed the `pedantic` configuration option to be `False` by default due to the vast proliferation of non-compliant VO Tables. [#296]
 - Renamed `astropy.io.vo` to `astropy.io.votable`.
- `astropy.table`
 - Added a workaround for an upstream bug in Numpy 1.6.2 that could cause a maximum recursion depth `RuntimeError` when printing table rows. [#341]
- `astropy.wcs`
 - Updated to `wcslib` 4.15 [#418]
 - Fixed a problem with handling FITS headers on locales that do not use dot as a decimal separator. This required an upstream fix to `wcslib` which is included in `wcslib` 4.14. [#313]
- Fixed some tests that could fail due to missing/incorrect logging configuration—ensures that tests don’t have any impact on the default log location or contents. [#291]
- Various minor documentation fixes [#293 and others]
- Fixed a bug where running the tests with the `py.test` command still tried to replace the system-installed `pytest` with the one bundled with Astropy. [#454]
- Improved multiprocessing compatibility for file downloads. [#615]
- Fixed handling of Cython modules when building from a source checkout of a tagged release version. [#594]
- Added a workaround for a bug in Sphinx that could occur when using the `:tocdepth:` directive. [#595]
- Minor `VOTable` fixes [#596]
- Fixed how `setup.py` uses `distribute_setup.py` to prevent possible `VersionConflict` errors when an older version of `distribute` is already installed on the user’s system. [#616][#640]
- Changed use of `log.warn` in the logging module to `log.warning` since the former is deprecated. [#624]

45.13 0.1 (2012-06-19)

- Initial release.

Part VI

Indices and Tables

- *genindex*
- *modindex*
- *search*

BIBLIOGRAPHY

- [R2] http://en.wikipedia.org/wiki/Great-circle_distance
- [R4] http://en.wikipedia.org/wiki/Great-circle_distance
- [R5] http://en.wikipedia.org/wiki/Airy_disk
- [R6] http://en.wikipedia.org/wiki/Gaussian_function
- [R7] <http://www.netlib.org/toms/733>
- [R8] Nelder, J.A. and Mead, R. (1965), "A simplex method for function minimization", *The Computer Journal*, 7, pp. 308-313
- [R9] David Shupe, et al, ADASS, ASP Conference Series, Vol. 347, 2005
- [R19] Calabretta, M.R., Greisen, E.W., 2002, *A&A*, 395, 1077 (Paper II)
- [R20] Calabretta, M.R., Greisen, E.W., 2002, *A&A*, 395, 1077 (Paper II)
- [R10] Brown, Lawrence D.; Cai, T. Tony; DasGupta, Anirban (2001). "Interval Estimation for a Binomial Proportion". *Statistical Science* 16 (2): 101-133. doi:10.1214/ss/1009213286
- [R11] Wilson, E. B. (1927). "Probable inference, the law of succession, and statistical inference". *Journal of the American Statistical Association* 22: 209-212.
- [R12] Jeffreys, Harold (1946). "An Invariant Form for the Prior Probability in Estimation Problems". *Proc. R. Soc. Lond.. A* 24 186 (1007): 453-461. doi:10.1098/rspa.1946.0056
- [R13] Jeffreys, Harold (1998). *Theory of Probability*. Oxford University Press, 3rd edition. ISBN 978-0198503682

a

- astropy.config, 953
- astropy.constants, 20
- astropy.convolution, 792
- astropy.coordinates, 256
- astropy.cosmology, 820
- astropy.io.ascii, 701
- astropy.io.fits, 593
- astropy.io.fits.diff, 674
- astropy.io.fits.scripts.fitscheck, 528
- astropy.io.fits.scripts.fitsheader, 528
- astropy.io.misc, 781
- astropy.io.misc.hdf5, 782
- astropy.io.registry, 960
- astropy.io.votable, 738
- astropy.io.votable.converters, 760
- astropy.io.votable.exceptions, 770
- astropy.io.votable.tree, 741
- astropy.io.votable.ucd, 763
- astropy.io.votable.util, 764
- astropy.io.votable.validator, 765
- astropy.io.votable.xmlutil, 766
- astropy.logger, 967
- astropy.modeling, 389
- astropy.modeling.fitting, 400
- astropy.modeling.functional_models, 408
- astropy.modeling.optimizers, 447
- astropy.modeling.polynomial, 457
- astropy.modeling.powerlaws, 450
- astropy.modeling.projections, 472
- astropy.modeling.rotations, 483
- astropy.modeling.statistic, 483
- astropy.nddata, 105
- astropy.stats, 855
- astropy.table, 156
- astropy.time, 201
- astropy.units, 53
- astropy.units.astrophys, 90
- astropy.units.cds, 92
- astropy.units.cgs, 89
- astropy.units.equivalencies, 95
- astropy.units.format, 83
- astropy.units.imperial, 91
- astropy.units.quantity, 43
- astropy.units.si, 88
- astropy.utils, 975
- astropy.utils.collections, 984
- astropy.utils.console, 985
- astropy.utils.data, 998
- astropy.utils.exceptions, 984
- astropy.utils.misc, 975
- astropy.utils.state, 996
- astropy.utils.timer, 991
- astropy.utils.xml.check, 1008
- astropy.utils.xml.iterparser, 1008
- astropy.utils.xml.unescaper, 1009
- astropy.utils.xml.validate, 1010
- astropy.utils.xml.writer, 1010
- astropy.vo, 867
- astropy.vo.client.async, 902
- astropy.vo.client.conesearch, 897
- astropy.vo.client.exceptions, 904
- astropy.vo.client.vos_catalog, 889
- astropy.vo.samp, 917
- astropy.vo.validator, 905
- astropy.vo.validator.exceptions, 908
- astropy.vo.validator.inspect, 906
- astropy.vo.validator.validate, 905
- astropy.wcs, 319
- astropy_helpers.sphinx.ext.automodapi, 1095
- astropy_helpers.sphinx.ext.automodsumm, 1096
- astropy_helpers.sphinx.ext.edit_on_github, 1097

Symbols

- `__call__()` (astropy.coordinates.CompositeTransform method), 273
- `__call__()` (astropy.coordinates.CoordinateTransform method), 274
- `__call__()` (astropy.coordinates.DynamicMatrixTransform method), 280
- `__call__()` (astropy.coordinates.FunctionTransform method), 288
- `__call__()` (astropy.coordinates.StaticMatrixTransform method), 302
- `__call__()` (astropy.io.ascii.BaseSplitter method), 712
- `__call__()` (astropy.io.ascii.DefaultSplitter method), 719
- `__call__()` (astropy.io.ascii.FixedWidthSplitter method), 723
- `__call__()` (astropy.io.ascii.TableOutputter method), 731
- `__call__()` (astropy.modeling.Fittable1DModel method), 390
- `__call__()` (astropy.modeling.Fittable2DModel method), 391
- `__call__()` (astropy.modeling.Model method), 396
- `__call__()` (astropy.modeling.SerialCompositeModel method), 400
- `__call__()` (astropy.modeling.SummedCompositeModel method), 400
- `__call__()` (astropy.modeling.fitting.Fitter method), 407
- `__call__()` (astropy.modeling.fitting.JointFitter method), 407
- `__call__()` (astropy.modeling.fitting.LevMarLSQFitter method), 403
- `__call__()` (astropy.modeling.fitting.LinearLSQFitter method), 401
- `__call__()` (astropy.modeling.fitting.SLSQPLSQFitter method), 404
- `__call__()` (astropy.modeling.fitting.SimplexLSQFitter method), 406
- `__call__()` (astropy.modeling.functional_models.Scale method), 440
- `__call__()` (astropy.modeling.functional_models.Shift method), 441
- `__call__()` (astropy.modeling.optimizers.Optimization method), 448
- `__call__()` (astropy.modeling.optimizers.SLSQP method), 449
- `__call__()` (astropy.modeling.optimizers.Simplex method), 450
- `__call__()` (astropy.modeling.polynomial.Chebyshev1D method), 458
- `__call__()` (astropy.modeling.polynomial.InverseSIP method), 461
- `__call__()` (astropy.modeling.polynomial.Legendre1D method), 462
- `__call__()` (astropy.modeling.polynomial.OrthoPolynomialBase method), 471
- `__call__()` (astropy.modeling.polynomial.Polynomial1D method), 465
- `__call__()` (astropy.modeling.polynomial.Polynomial2D method), 467
- `__call__()` (astropy.modeling.polynomial.SIP method), 469
- `__call__()` (astropy.modeling.projections.AffineTransformation2D method), 482
- `__call__()` (astropy.modeling.projections.Pix2Sky_AZP method), 473
- `__call__()` (astropy.modeling.projections.Pix2Sky_CAR method), 474
- `__call__()` (astropy.modeling.projections.Pix2Sky_CEA method), 476
- `__call__()` (astropy.modeling.projections.Pix2Sky_CYP method), 477
- `__call__()` (astropy.modeling.projections.Pix2Sky_MER method), 478
- `__call__()` (astropy.modeling.projections.Pix2Sky_SIN method), 479
- `__call__()` (astropy.modeling.projections.Pix2Sky_STG method), 480
- `__call__()` (astropy.modeling.projections.Pix2Sky_TAN method), 481
- `__call__()` (astropy.modeling.projections.Sky2Pix_AZP method), 474
- `__call__()` (astropy.modeling.projections.Sky2Pix_CAR method), 475
- `__call__()` (astropy.modeling.projections.Sky2Pix_CEA method), 476

- `__call__()` (astropy.modeling.projections.Sky2Pix_CYP method), 478
 - `__call__()` (astropy.modeling.projections.Sky2Pix_MER method), 479
 - `__call__()` (astropy.modeling.projections.Sky2Pix_SIN method), 479
 - `__call__()` (astropy.modeling.projections.Sky2Pix_STG method), 480
 - `__call__()` (astropy.modeling.projections.Sky2Pix_TAN method), 481
 - `__call__()` (astropy.modeling.rotations.RotateCelestial2Native method), 484
 - `__call__()` (astropy.modeling.rotations.RotateNative2Celestial method), 485
 - `__call__()` (astropy.modeling.rotations.Rotation2D method), 485
 - `__call__()` (astropy.vo.samp.SAMPMsgReplierWrapper method), 941
- ## A
- `a` (astropy.wcs.Sip attribute), 325
 - `a_order` (astropy.wcs.Sip attribute), 325
 - AASTex (class in astropy.io.ascii), 705
 - `abbrev` (astropy.constants.Constant attribute), 21
 - `absorption_distance()` (astropy.cosmology.FLRW method), 832
 - `acc` (astropy.modeling.optimizers.Optimization attribute), 448
 - `add()` (astropy.modeling.LabeledInput method), 393
 - `add()` (astropy.nddata.NDData method), 108
 - `add_blank()` (astropy.io.fits.Header method), 620
 - `add_catalog()` (astropy.vo.client.vos_catalog.VOSDatabase method), 894
 - `add_catalog_by_url()` (astropy.vo.client.vos_catalog.VOSDatabase method), 894
 - `add_checksum()` (astropy.io.fits.BinTableHDU method), 634
 - `add_checksum()` (astropy.io.fits.CompImageHDU method), 664
 - `add_checksum()` (astropy.io.fits.GroupsHDU method), 611
 - `add_checksum()` (astropy.io.fits.ImageHDU method), 656
 - `add_checksum()` (astropy.io.fits.PrimaryHDU method), 604
 - `add_checksum()` (astropy.io.fits.TableHDU method), 643
 - `add_col()` (astropy.io.fits.ColDefs method), 650
 - `add_column()` (astropy.table.Table method), 172
 - `add_columns()` (astropy.table.Table method), 172
 - `add_comment()` (astropy.io.fits.Header method), 620
 - `add_datasum()` (astropy.io.fits.BinTableHDU method), 635
 - `add_datasum()` (astropy.io.fits.CompImageHDU method), 665
 - `add_datasum()` (astropy.io.fits.GroupsHDU method), 611
 - `add_datasum()` (astropy.io.fits.ImageHDU method), 657
 - `add_datasum()` (astropy.io.fits.PrimaryHDU method), 605
 - `add_datasum()` (astropy.io.fits.TableHDU method), 643
 - `add_enabled_equivalencies()` (in module astropy.units), 54
 - `add_enabled_units()` (in module astropy.units), 54
 - `add_history()` (astropy.io.fits.Header method), 620
 - `add_model()` (astropy.modeling.Model method), 396
 - `add_row()` (astropy.table.Table method), 173
 - `add_transform()` (astropy.coordinates.TransformGraph method), 304
 - AffineTransformation2D (class in astropy.modeling.projections), 481
 - `age()` (astropy.cosmology.FLRW method), 832
 - `aggregate()` (astropy.table.ColumnGroups method), 164
 - `aggregate()` (astropy.table.TableGroups method), 187
 - AiryDisk2D (class in astropy.modeling.functional_models), 409
 - AiryDisk2DKernel (class in astropy.convolution), 797
 - `aliases` (astropy.units.NamedUnit attribute), 65
 - `aliases` (astropy.units.UnitBase attribute), 78
 - `all()` (astropy.units.Quantity method), 68
 - `all()` (astropy.units.quantity.Quantity method), 46
 - `all_pix2world()` (astropy.wcs.WCS method), 331
 - `all_world2pix()` (astropy.wcs.WCS method), 332
 - AllType (class in astropy.io.ascii), 705
 - `alpha` (astropy.modeling.functional_models.Beta1D attribute), 413
 - `alpha` (astropy.modeling.functional_models.Beta2D attribute), 415
 - `alpha` (astropy.modeling.powerlaws.ExponentialCutoffPowerLaw1D attribute), 453
 - `alpha` (astropy.modeling.powerlaws.LogParabola1D attribute), 454
 - `alpha` (astropy.modeling.powerlaws.PowerLaw1D attribute), 456
 - `alpha_1` (astropy.modeling.powerlaws.BrokenPowerLaw1D attribute), 452
 - `alpha_2` (astropy.modeling.powerlaws.BrokenPowerLaw1D attribute), 452
 - `alt` (astropy.wcs.Wcsprm attribute), 348
 - AltAz (class in astropy.coordinates), 260
 - `amplitude` (astropy.modeling.functional_models.AiryDisk2D attribute), 411
 - `amplitude` (astropy.modeling.functional_models.Beta1D attribute), 413
 - `amplitude` (astropy.modeling.functional_models.Beta2D attribute), 415
 - `amplitude` (astropy.modeling.functional_models.Box1D attribute), 416
 - `amplitude` (astropy.modeling.functional_models.Box2D attribute), 418

- amplitude (astropy.modeling.functional_models.Const1D attribute), 420
- amplitude (astropy.modeling.functional_models.Const2D attribute), 421
- amplitude (astropy.modeling.functional_models.Disk2D attribute), 423
- amplitude (astropy.modeling.functional_models.Gaussian1D attribute), 425
- amplitude (astropy.modeling.functional_models.Gaussian2D attribute), 428
- amplitude (astropy.modeling.functional_models.GaussianBeam attribute), 430
- amplitude (astropy.modeling.functional_models.Lorentz1D attribute), 433
- amplitude (astropy.modeling.functional_models.MexicanHat1D attribute), 435
- amplitude (astropy.modeling.functional_models.MexicanHat2D attribute), 437
- amplitude (astropy.modeling.functional_models.Ring2D attribute), 439
- amplitude (astropy.modeling.functional_models.Sine1D attribute), 443
- amplitude (astropy.modeling.functional_models.Trapezoid1D attribute), 444
- amplitude (astropy.modeling.functional_models.TrapezoidDisk2D attribute), 447
- amplitude (astropy.modeling.powerlaws.BrokenPowerLaw1D attribute), 452
- amplitude (astropy.modeling.powerlaws.ExponentialCutoffPowerLaw1D attribute), 453
- amplitude (astropy.modeling.powerlaws.LogParabola1D attribute), 454
- amplitude (astropy.modeling.powerlaws.PowerLaw1D attribute), 456
- angle (astropy.modeling.rotations.Rotation2D attribute), 485
- Angle (class in astropy.coordinates), 261
- angular_diameter_distance() (astropy.cosmology.FLRW method), 832
- angular_diameter_distance() (in module astropy.cosmology), 821
- angular_diameter_distance_z1z2() (astropy.cosmology.FLRW method), 832
- any() (astropy.units.Quantity method), 68
- any() (astropy.units.quantity.Quantity method), 46
- ap (astropy.wcs.Sip attribute), 325
- ap_order (astropy.wcs.Sip attribute), 325
- append() (astropy.io.fits.CardList method), 631
- append() (astropy.io.fits.HDUList method), 601
- append() (astropy.io.fits.Header method), 620
- append() (astropy.utils.collections.HomogeneousList method), 985
- append() (in module astropy.io.fits), 596
- arcsec_per_kpc_comoving() (astropy.cosmology.FLRW method), 833
- arcsec_per_kpc_comoving() (in module astropy.cosmology), 821
- arcsec_per_kpc_proper() (astropy.cosmology.FLRW method), 833
- arcsec_per_kpc_proper() (in module astropy.cosmology), 822
- argmax() (astropy.units.Quantity method), 68
- argmax() (astropy.units.quantity.Quantity method), 46
- argmin() (astropy.units.Quantity method), 68
- argmin() (astropy.units.quantity.Quantity method), 46
- argsort() (astropy.table.Table method), 174
- argsort() (astropy.units.Quantity method), 68
- argsort() (astropy.units.quantity.Quantity method), 46
- array (astropy.convolution.CustomKernel attribute), 801
- array (astropy.convolution.Kernel attribute), 804
- array (astropy.nddata.StdDevUncertainty attribute), 113
- arraysize (astropy.io.votable.tree.Field attribute), 746
- ascard (astropy.io.fits.Header attribute), 621
- ascardimage() (astropy.io.fits.Card method), 630
- astropy.config (module), 953
- astropy.constants (module), 20
- astropy.convolution (module), 792
- astropy.coordinates (module), 256
- astropy.cosmology (module), 820
- astropy.io.ascii (module), 701
- astropy.io.fits (module), 593
- astropy.io.fits.diff (module), 674
- astropy.io.fits.scripts.fitscheck (module), 528
- astropy.io.fits.scripts.fitsheader (module), 528
- astropy.io.misc (module), 781
- astropy.io.misc.hdf5 (module), 782
- astropy.io.registry (module), 960
- astropy.io.votable (module), 738
- astropy.io.votable.converters (module), 760
- astropy.io.votable.exceptions (module), 770
- astropy.io.votable.tree (module), 741
- astropy.io.votable.ucd (module), 763
- astropy.io.votable.util (module), 764
- astropy.io.votable.validator (module), 765
- astropy.io.votable.xmlutil (module), 766
- astropy.logger (module), 967
- astropy.modeling (module), 389
- astropy.modeling.fitting (module), 400
- astropy.modeling.functional_models (module), 408
- astropy.modeling.optimizers (module), 447
- astropy.modeling.polynomial (module), 457
- astropy.modeling.powerlaws (module), 450
- astropy.modeling.projections (module), 472
- astropy.modeling.rotations (module), 483
- astropy.modeling.statistic (module), 483
- astropy.nddata (module), 105
- astropy.stats (module), 855

- astropy.table (module), 156
 - astropy.time (module), 201
 - astropy.units (module), 53
 - astropy.units.astrophys (module), 90
 - astropy.units.cds (module), 92
 - astropy.units.cgs (module), 89
 - astropy.units.equivalencies (module), 95
 - astropy.units.format (module), 83
 - astropy.units.imperial (module), 91
 - astropy.units.quantity (module), 43
 - astropy.units.si (module), 88
 - astropy.utils (module), 975
 - astropy.utils.collections (module), 984
 - astropy.utils.console (module), 985
 - astropy.utils.data (module), 998
 - astropy.utils.exceptions (module), 984
 - astropy.utils.misc (module), 975
 - astropy.utils.state (module), 996
 - astropy.utils.timer (module), 991
 - astropy.utils.xml.check (module), 1008
 - astropy.utils.xml.iterparser (module), 1008
 - astropy.utils.xml.unescaper (module), 1009
 - astropy.utils.xml.validate (module), 1010
 - astropy.utils.xml.writer (module), 1010
 - astropy.vo (module), 867, 888
 - astropy.vo.client.async (module), 902
 - astropy.vo.client.conesearch (module), 897
 - astropy.vo.client.exceptions (module), 904
 - astropy.vo.client.vos_catalog (module), 889
 - astropy.vo.samp (module), 917
 - astropy.vo.validator (module), 905
 - astropy.vo.validator.exceptions (module), 908
 - astropy.vo.validator.inspect (module), 906
 - astropy.vo.validator.validate (module), 905
 - astropy.wcs (module), 319
 - astropy_helpers.sphinx.ext.automodapi (module), 1095
 - astropy_helpers.sphinx.ext.automodsumm (module), 1096
 - astropy_helpers.sphinx.ext.edit_on_github (module), 1097
 - AstropyBackwardsIncompatibleChangeWarning, 984
 - AstropyDeprecationWarning, 984
 - AstropyLogger (class in astropy.logger), 968
 - AstropyPendingDeprecationWarning, 984
 - AstropyUserWarning, 984
 - AstropyWarning, 984
 - AsyncBase (class in astropy.vo.client.async), 903
 - AsyncConeSearch (class in astropy.vo.client.conesearch), 901
 - AsyncSearchAll (class in astropy.vo.client.conesearch), 902
 - attr_classes (astropy.coordinates.CartesianRepresentation attribute), 272
 - attr_classes (astropy.coordinates.CylindricalRepresentation attribute), 275
 - attr_classes (astropy.coordinates.PhysicsSphericalRepresentation attribute), 294
 - attr_classes (astropy.coordinates.SphericalRepresentation attribute), 301
 - attr_classes (astropy.coordinates.UnitSphericalRepresentation attribute), 308
 - auto_colname (astropy.table.Conf attribute), 164
 - auto_format (astropy.io.ascii.BaseHeader attribute), 707
 - axis_type_names (astropy.wcs.WCS attribute), 331
 - axis_types (astropy.wcs.Wcsprm attribute), 348
- ## B
- b (astropy.wcs.Sip attribute), 325
 - b_order (astropy.wcs.Sip attribute), 325
 - Base (class in astropy.units.format), 84
 - BaseCoordinateFrame (class in astropy.coordinates), 265
 - BaseData (class in astropy.io.ascii), 705
 - BaseHeader (class in astropy.io.ascii), 707
 - BaseInputter (class in astropy.io.ascii), 708
 - BaseOutputter (class in astropy.io.ascii), 709
 - BaseReader (class in astropy.io.ascii), 709
 - BaseRepresentation (class in astropy.coordinates), 269
 - bases (astropy.units.CompositeUnit attribute), 62
 - bases (astropy.units.UnitBase attribute), 78
 - BaseSplitter (class in astropy.io.ascii), 711
 - BaseVOError, 904
 - BaseVOValidationError, 908
 - Basic (class in astropy.io.ascii), 712
 - beta (astropy.modeling.powerlaws.LogParabola1D attribute), 454
 - Beta1D (class in astropy.modeling.functional_models), 411
 - Beta2D (class in astropy.modeling.functional_models), 413
 - bind_receive_call() (astropy.vo.samp.SAMPClient method), 920
 - bind_receive_call() (astropy.vo.samp.SAMPIntegratedClient method), 932
 - bind_receive_message() (astropy.vo.samp.SAMPClient method), 920
 - bind_receive_message() (astropy.vo.samp.SAMPIntegratedClient method), 933
 - bind_receive_notification() (astropy.vo.samp.SAMPClient method), 921
 - bind_receive_notification() (astropy.vo.samp.SAMPIntegratedClient method), 933
 - bind_receive_response() (astropy.vo.samp.SAMPClient method), 921

- [bind_receive_response\(\)](#) (astropy.vo.samp.SAMPIntegratedClient method), 934
[binned_binom_proportion\(\)](#) (in module astropy.stats), 856
[binom_conf_interval\(\)](#) (in module astropy.stats), 857
[binoutput\(\)](#) (astropy.io.votable.converters.Converter method), 762
[binparse\(\)](#) (astropy.io.votable.converters.Converter method), 762
[BinTableHDU](#) (class in astropy.io.fits), 634
[biweight_location\(\)](#) (in module astropy.stats), 859
[biweight_midvariance\(\)](#) (in module astropy.stats), 860
[bookend](#) (astropy.io.ascii.FixedWidthSplitter attribute), 723
[bootstrap\(\)](#) (in module astropy.stats), 861
[bounds](#) (astropy.modeling.Model attribute), 395
[bounds](#) (astropy.modeling.Parameter attribute), 398
[bounds_check\(\)](#) (astropy.wcs.Wcsprm method), 354
[BoundsError](#), 270
[Box1D](#) (class in astropy.modeling.functional_models), 415
[Box1DKernel](#) (class in astropy.convolution), 798
[Box2D](#) (class in astropy.modeling.functional_models), 417
[Box2DKernel](#) (class in astropy.convolution), 799
[bp](#) (astropy.wcs.Sip attribute), 325
[bp_order](#) (astropy.wcs.Sip attribute), 325
[brightness_temperature\(\)](#) (in module astropy.units), 55
[brightness_temperature\(\)](#) (in module astropy.units.equivalencies), 98
[BrokenPowerLaw1D](#) (class in astropy.modeling.powerlaws), 451
- ## C
- [CacheMissingWarning](#), 1008
[calc_footprint\(\)](#) (astropy.wcs.WCS method), 334
[calcFootprint\(\)](#) (astropy.wcs.WCS method), 334
[call\(\)](#) (astropy.vo.samp.SAMPProxy method), 925
[call\(\)](#) (astropy.vo.samp.SAMPIntegratedClient method), 934
[call_all\(\)](#) (astropy.vo.samp.SAMPProxy method), 925
[call_all\(\)](#) (astropy.vo.samp.SAMPIntegratedClient method), 934
[call_and_wait\(\)](#) (astropy.vo.samp.SAMPProxy method), 926
[call_and_wait\(\)](#) (astropy.vo.samp.SAMPIntegratedClient method), 934
[call_vo_service\(\)](#) (in module astropy.vo.client.vos_catalog), 890
[Card](#) (class in astropy.io.fits), 630
[cardimage](#) (astropy.io.fits.Card attribute), 630
[CardList](#) (class in astropy.io.fits), 631
[cards](#) (astropy.io.fits.Header attribute), 621
[cartesian](#) (astropy.coordinates.BaseCoordinateFrame attribute), 266
[cartesian_to_spherical\(\)](#) (in module astropy.coordinates), 256
[CartesianPoints](#) (class in astropy.coordinates), 271
[CartesianRepresentation](#) (class in astropy.coordinates), 271
[cd](#) (astropy.wcs.Wcsprm attribute), 349
[cdelt](#) (astropy.wcs.DistortionLookupTable attribute), 322
[cdelt](#) (astropy.wcs.Wcsprm attribute), 349
[cdfix\(\)](#) (astropy.wcs.Wcsprm method), 355
[Cds](#) (class in astropy.io.ascii), 713
[CDS](#) (class in astropy.units.format), 85
[cel_offset](#) (astropy.wcs.Wcsprm attribute), 349
[celfix\(\)](#) (astropy.wcs.Wcsprm method), 355
[center](#) (astropy.convolution.Kernel attribute), 804
[cgs](#) (astropy.constants.Constant attribute), 21
[cgs](#) (astropy.constants.EMConstant attribute), 22
[cgs](#) (astropy.units.Quantity attribute), 67
[cgs](#) (astropy.units.quantity.Quantity attribute), 45
[cgs](#) (astropy.units.UnitBase attribute), 78
[change_attrib\(\)](#) (astropy.io.fits.ColDefs method), 650
[change_name\(\)](#) (astropy.io.fits.ColDefs method), 650
[change_unit\(\)](#) (astropy.io.fits.ColDefs method), 650
[Chebyshev1D](#) (class in astropy.modeling.polynomial), 457
[Chebyshev2D](#) (class in astropy.modeling.polynomial), 459
[check_anyuri\(\)](#) (in module astropy.io.votable.xmlutil), 767
[check_anyuri\(\)](#) (in module astropy.utils.xml.check), 1008
[check_conesearch_sites\(\)](#) (in module astropy.vo.validator.validate), 905
[check_free_space_in_dir\(\)](#) (in module astropy.utils.data), 1005
[check_id\(\)](#) (in module astropy.io.votable.xmlutil), 766
[check_id\(\)](#) (in module astropy.utils.xml.check), 1008
[check_mime_content_type\(\)](#) (in module astropy.io.votable.xmlutil), 767
[check_mime_content_type\(\)](#) (in module astropy.utils.xml.check), 1008
[check_mu\(\)](#) (astropy.modeling.projections.Pix2Sky_AZP method), 473
[check_mu\(\)](#) (astropy.modeling.projections.Sky2Pix_AZP method), 474
[check_token\(\)](#) (in module astropy.io.votable.xmlutil), 767
[check_token\(\)](#) (in module astropy.utils.xml.check), 1008
[check_ucd\(\)](#) (in module astropy.io.votable.ucd), 764
[choose\(\)](#) (astropy.units.Quantity method), 69
[choose\(\)](#) (astropy.units.quantity.Quantity method), 47
[clear\(\)](#) (astropy.io.fits.Header method), 621
[clear_download_cache\(\)](#) (in module astropy.utils.data), 1005

- clenshaw() (astropy.modeling.polynomial.Chebyshev1D method), 458
 clenshaw() (astropy.modeling.polynomial.Legendre1D method), 462
 clip() (astropy.units.Quantity method), 69
 clip() (astropy.units.quantity.Quantity method), 47
 close() (astropy.io.fits.HDUList method), 601
 close() (astropy.io.fits.StreamingHDU method), 618
 close() (astropy.utils.xml.writer.XMLWriter method), 1011
 cname (astropy.wcs.Wcsprm attribute), 349
 coerce_range_list_param() (in module astropy.io.votable.util), 765
 col_fit_deriv (astropy.modeling.FittableModel attribute), 392
 colax (astropy.wcs.Wcsprm attribute), 349
 ColDefs (class in astropy.io.fits), 649
 colnames (astropy.io.ascii.BaseHeader attribute), 707
 colnames (astropy.table.Row attribute), 169
 colnames (astropy.table.Table attribute), 171
 colnum (astropy.wcs.Wcsprm attribute), 349
 color_print() (in module astropy.utils.console), 986
 Column (class in astropy.io.ascii), 714
 Column (class in astropy.io.fits), 648
 Column (class in astropy.table), 160
 ColumnClass (astropy.table.Table attribute), 171
 ColumnGroups (class in astropy.table), 163
 columns (astropy.io.fits.BinTableHDU attribute), 635
 columns (astropy.io.fits.CompImageHDU attribute), 666
 columns (astropy.io.fits.FITS_rec attribute), 651
 columns (astropy.io.fits.TableHDU attribute), 644
 columns (astropy.table.Row attribute), 169
 comment (astropy.io.ascii.BaseData attribute), 706
 comment (astropy.io.ascii.BaseHeader attribute), 707
 comment (astropy.io.fits.Card attribute), 630
 comment() (astropy.utils.xml.writer.XMLWriter method), 1011
 comment_lines (astropy.io.ascii.BaseReader attribute), 710
 CommentedHeader (class in astropy.io.ascii), 715
 comments (astropy.io.fits.Header attribute), 621
 comoving_distance() (astropy.cosmology.FLRW method), 833
 comoving_distance() (in module astropy.cosmology), 822
 comoving_transverse_distance() (astropy.cosmology.FLRW method), 833
 comoving_volume() (astropy.cosmology.FLRW method), 834
 compare() (astropy.wcs.Wcsprm method), 355
 compData (astropy.io.fits.CompImageHDU attribute), 666
 CompImageHDU (class in astropy.io.fits), 662
 components (astropy.coordinates.BaseRepresentation attribute), 270
 compose() (astropy.units.UnitBase method), 78
 CompositeTransform (class in astropy.coordinates), 272
 CompositeUnit (class in astropy.units), 62
 compute_hash() (in module astropy.utils.data), 1004
 compute_hash_block_size (astropy.utils.data.Conf attribute), 1007
 compute_z() (astropy.coordinates.Distance method), 278
 conesearch() (in module astropy.vo.client.conesearch), 897
 conesearch_dbname (astropy.vo.Conf attribute), 889
 conesearch_master_list (astropy.vo.validator.Conf attribute), 905
 conesearch_timer() (in module astropy.vo.client.conesearch), 901
 conesearch_urls (astropy.vo.validator.Conf attribute), 905
 ConeSearchError, 904
 ConeSearchResults (class in astropy.vo.validator.inspect), 906
 Conf (class in astropy.io.votable), 741
 Conf (class in astropy.logger), 967
 Conf (class in astropy.nddata), 105
 Conf (class in astropy.table), 164
 Conf (class in astropy.utils.data), 1007
 Conf (class in astropy.vo), 889
 Conf (class in astropy.vo.samp), 917
 Conf (class in astropy.vo.validator), 905
 ConfigAlias (astropy.config attribute), 955
 ConfigItem (astropy.config attribute), 956
 ConfigNamespace (class in astropy.config), 956
 ConfigurationItem (astropy.config attribute), 958
 ConfigurationMissingWarning, 958
 connect() (astropy.vo.samp.SAMPHubProxy method), 926
 connect() (astropy.vo.samp.SAMPIntegratedClient method), 934
 consent() (astropy.vo.samp.WebProfileDialog method), 942
 Console (class in astropy.units.format), 85
 Const1D (class in astropy.modeling.functional_models), 419
 Const2D (class in astropy.modeling.functional_models), 420
 Constant (class in astropy.constants), 21
 content (astropy.io.votable.tree.Info attribute), 743
 content_role (astropy.io.votable.tree.Link attribute), 742
 content_type (astropy.io.votable.tree.Link attribute), 742
 continuation_char (astropy.io.ascii.ContinuationLinesInputter attribute), 715
 ContinuationLinesInputter (class in astropy.io.ascii), 715
 convert_bytestring_to_unicode() (astropy.table.Table method), 174
 convert_input() (astropy.coordinates.FrameAttribute method), 287

- `convert_input()` (`astropy.coordinates.TimeFrameAttribute` method), 303
`convert_numpy()` (in module `astropy.io.ascii`), 701
`convert_to_writable_filelike()` (in module `astropy.io.votable.util`), 764
`convert_unicode_to_bytestring()` (`astropy.table.Table` method), 175
`convert_unit_to()` (`astropy.nddata.NDData` method), 109
`convert_unit_to()` (`astropy.table.Column` method), 161
`convert_unit_to()` (`astropy.table.MaskedColumn` method), 167
`Converter` (class in `astropy.io.votable.converters`), 761
`ConvertError`, 273
`converters` (`astropy.io.ascii.BaseOutputter` attribute), 709
`convolve()` (in module `astropy.convolution`), 792
`convolve_fft()` (in module `astropy.convolution`), 793
`coord` (`astropy.wcs.Tabprm` attribute), 327
`coordinate_systems` (`astropy.io.votable.tree.Resource` attribute), 756
`coordinate_systems` (`astropy.io.votable.tree.VOTableFile` attribute), 758
`CoordinateTransform` (class in `astropy.coordinates`), 273
`CooSys` (class in `astropy.io.votable.tree`), 748
`copy()` (`astropy.constants.Constant` method), 22
`copy()` (`astropy.io.fits.BinTableHDU` method), 635
`copy()` (`astropy.io.fits.CardList` method), 632
`copy()` (`astropy.io.fits.Column` method), 649
`copy()` (`astropy.io.fits.CompImageHDU` method), 666
`copy()` (`astropy.io.fits.FITS_rec` method), 651
`copy()` (`astropy.io.fits.GroupsHDU` method), 612
`copy()` (`astropy.io.fits.Header` method), 621
`copy()` (`astropy.io.fits.ImageHDU` method), 657
`copy()` (`astropy.io.fits.PrimaryHDU` method), 605
`copy()` (`astropy.io.fits.TableHDU` method), 644
`copy()` (`astropy.modeling.LabeledInput` method), 393
`copy()` (`astropy.modeling.Model` method), 396
`copy()` (`astropy.table.Column` method), 161
`copy()` (`astropy.table.MaskedColumn` method), 167
`copy()` (`astropy.table.Table` method), 175
`copy()` (`astropy.time.Time` method), 204
`copy()` (`astropy.wcs.WCS` method), 334
`count()` (`astropy.io.fits.CardList` method), 632
`count()` (`astropy.io.fits.Header` method), 622
`count_blanks()` (`astropy.io.fits.CardList` method), 632
`cpdis1` (`astropy.wcs.WCSBase` attribute), 345
`cpdis2` (`astropy.wcs.WCSBase` attribute), 345
`crder` (`astropy.wcs.Wcsprm` attribute), 350
`create()` (`astropy.vo.client.vos_catalog.VOSCatalog` class method), 892
`create_arrays()` (`astropy.io.votable.tree.Table` method), 754
`create_card()` (in module `astropy.io.fits`), 633
`create_card_from_string()` (in module `astropy.io.fits`), 633
`create_empty()` (`astropy.vo.client.vos_catalog.VOSDatabase` class method), 894
`create_mask()` (`astropy.table.Table` method), 175
`critical_density()` (`astropy.cosmology.FLRW` method), 834
`critical_density()` (in module `astropy.cosmology`), 822
`critical_density0` (`astropy.cosmology.FLRW` attribute), 830
`crota` (`astropy.wcs.Wcsprm` attribute), 350
`crpix` (`astropy.wcs.DistortionLookupTable` attribute), 322
`crpix` (`astropy.wcs.Sip` attribute), 325
`crpix` (`astropy.wcs.Wcsprm` attribute), 350
`crval` (`astropy.wcs.DistortionLookupTable` attribute), 322
`crval` (`astropy.wcs.Tabprm` attribute), 327
`crval` (`astropy.wcs.Wcsprm` attribute), 350
`Csv` (class in `astropy.io.ascii`), 716
`csyer` (`astropy.wcs.Wcsprm` attribute), 350
`ctype` (`astropy.wcs.Wcsprm` attribute), 350
`cubeface` (`astropy.wcs.Wcsprm` attribute), 350
`cumprod()` (`astropy.units.Quantity` method), 69
`cumprod()` (`astropy.units.quantity.Quantity` method), 47
`cumsum()` (`astropy.units.Quantity` method), 69
`cumsum()` (`astropy.units.quantity.Quantity` method), 47
`cunit` (`astropy.wcs.Wcsprm` attribute), 351
`custom_model_id()` (in module `astropy.modeling.functional_models`), 408
`CustomKernel` (class in `astropy.convolution`), 800
`cylfix()` (`astropy.wcs.Wcsprm` method), 355
`CylindricalRepresentation` (class in `astropy.coordinates`), 275
- ## D
- `Daophot` (class in `astropy.io.ascii`), 717
`data` (`astropy.coordinates.BaseCoordinateFrame` attribute), 266
`data` (`astropy.io.fits.GroupData` attribute), 617
`data` (`astropy.io.fits.GroupsHDU` attribute), 612
`data` (`astropy.io.fits.ImageHDU` attribute), 657
`data` (`astropy.io.fits.PrimaryHDU` attribute), 605
`data` (`astropy.table.MaskedColumn` attribute), 166
`data` (`astropy.table.Row` attribute), 169
`data` (`astropy.wcs.DistortionLookupTable` attribute), 322
`data()` (`astropy.utils.xml.writer.XMLWriter` method), 1012
`datatype` (`astropy.io.votable.tree.Field` attribute), 746
`dataurl` (`astropy.utils.data.Conf` attribute), 1007
`dateavg` (`astropy.wcs.Wcsprm` attribute), 351
`dateobs` (`astropy.wcs.Wcsprm` attribute), 351
`datfix()` (`astropy.wcs.Wcsprm` method), 355
`de_density_scale()` (`astropy.cosmology.FLRW` method), 834
`de_density_scale()` (`astropy.cosmology.LambdaCDM` method), 843

- `de_density_scale()` (astropy.cosmology.w0waCDM method), 846
`de_density_scale()` (astropy.cosmology.w0wzCDM method), 849
`de_density_scale()` (astropy.cosmology.wCDM method), 851
`de_density_scale()` (astropy.cosmology.wpwaCDM method), 853
`declare_metadata()` (astropy.vo.samp.SAMPClient method), 922
`declare_metadata()` (astropy.vo.samp.SAMPHubProxy method), 926
`declare_metadata()` (astropy.vo.samp.SAMPIntegratedClient method), 935
`declare_subscriptions()` (astropy.vo.samp.SAMPClient method), 922
`declare_subscriptions()` (astropy.vo.samp.SAMPHubProxy method), 926
`declare_subscriptions()` (astropy.vo.samp.SAMPIntegratedClient method), 935
`decompose()` (astropy.units.CompositeUnit method), 63
`decompose()` (astropy.units.IrreducibleUnit method), 63
`decompose()` (astropy.units.Quantity method), 69
`decompose()` (astropy.units.quantity.Quantity method), 47
`decompose()` (astropy.units.Unit method), 77
`decompose()` (astropy.units.UnitBase method), 79
`deepcopy()` (astropy.wcs.WCS method), 334
`def_physical_type()` (in module astropy.units), 56
`def_unit()` (in module astropy.units), 56
`default` (astropy.modeling.Parameter attribute), 398
`default()` (astropy.utils.misc.JsonCustomEncoder method), 983
`default_converters` (astropy.io.ascii.TableOutputter attribute), 731
`default_cosmology` (class in astropy.cosmology), 844
`default_representation` (astropy.coordinates.AltAz attribute), 261
`default_representation` (astropy.coordinates.BaseCoordinateFrame attribute), 266
`default_representation` (astropy.coordinates.FK4 attribute), 284
`default_representation` (astropy.coordinates.FK4NoETerms attribute), 285
`default_representation` (astropy.coordinates.FK5 attribute), 286
`default_representation` (astropy.coordinates.Galactic attribute), 289
`default_representation` (astropy.coordinates.GenericFrame attribute), 289
`default_representation` (astropy.coordinates.ICRS attribute), 290
`DefaultSplitter` (class in astropy.io.ascii), 718
`degree` (astropy.modeling.polynomial.PolynomialModel attribute), 471
`del_col()` (astropy.io.fits.ColDefs method), 650
`delete_attribute()` (astropy.vo.client.vos_catalog.VOSCatalog method), 892
`delete_catalog()` (astropy.vo.client.vos_catalog.VOSDatabase method), 895
`delete_catalog_by_url()` (astropy.vo.client.vos_catalog.VOSDatabase method), 895
`delete_temporary_downloads_at_exit` (astropy.utils.data.Conf attribute), 1007
`deleter()` (astropy.utils.misc.lazyproperty method), 981
`delimiter` (astropy.io.ascii.BaseSplitter attribute), 712
`delimiter` (astropy.io.ascii.DefaultSplitter attribute), 719
`delimiter_pad` (astropy.io.ascii.FixedWidthSplitter attribute), 723
`delta` (astropy.wcs.Tabprm attribute), 327
`delta_tdb_tt` (astropy.time.Time attribute), 203
`delta_ut1_utc` (astropy.time.Time attribute), 203
`delval()` (in module astropy.io.fits), 600
`deprecated()` (in module astropy.utils.misc), 977
`deprecated_attribute()` (in module astropy.utils.misc), 978
`deregister()` (astropy.units.NamedUnit method), 65
`det2im()` (astropy.wcs.WCS method), 334
`det2im1` (astropy.wcs.WCSBase attribute), 345
`det2im2` (astropy.wcs.WCSBase attribute), 345
`diff()` (astropy.units.Quantity method), 70
`diff()` (astropy.units.quantity.Quantity method), 48
`differential_comoving_volume()` (astropy.cosmology.FLRW method), 835
`dimension` (astropy.convolution.Kernel attribute), 804
`dimensionless_angles()` (in module astropy.units), 57
`dimensionless_angles()` (in module astropy.units.equivalencies), 98
`disable_color()` (astropy.logger.AstropyLogger method), 969
`disable_exception_logging()` (astropy.logger.AstropyLogger method), 969
`disable_warnings_logging()` (astropy.logger.AstropyLogger method), 969
`disconnect()` (astropy.vo.samp.SAMPHubProxy method), 926
`disconnect()` (astropy.vo.samp.SAMPIntegratedClient method), 935
`discretize_model()` (in module astropy.convolution), 795
`Disk2D` (class in astropy.modeling.functional_models), 422
`distance` (astropy.coordinates.SphericalRepresentation attribute), 301

- Distance (class in `astropy.coordinates`), 276
- `distmod` (`astropy.coordinates.Distance` attribute), 277
- `distmod` (`astropy.cosmology.FLRW` method), 835
- `distmod` (in module `astropy.cosmology`), 823
- `DistortionLookupTable` (class in `astropy.wcs`), 321
- `divide` (`astropy.nddata.NDData` method), 109
- `dms` (`astropy.coordinates.Angle` attribute), 263
- `do_fit` (`astropy.utils.timer.RunTimePredictor` method), 995
- `doppler_optical` (in module `astropy.units`), 57
- `doppler_optical` (in module `astropy.units.equivalencies`), 96
- `doppler_radio` (in module `astropy.units`), 58
- `doppler_radio` (in module `astropy.units.equivalencies`), 96
- `doppler_relativistic` (in module `astropy.units`), 58
- `doppler_relativistic` (in module `astropy.units.equivalencies`), 97
- `dot` (`astropy.units.Quantity` method), 70
- `dot` (`astropy.units.quantity.Quantity` method), 48
- `doublequote` (`astropy.io.ascii.DefaultSplitter` attribute), 719
- `download_block_size` (`astropy.utils.data.Conf` attribute), 1007
- `download_cache_lock_attempts` (`astropy.utils.data.Conf` attribute), 1007
- `download_file` (in module `astropy.utils.data`), 1006
- `download_files_in_parallel` (in module `astropy.utils.data`), 1006
- `dropaxis` (`astropy.wcs.WCS` method), 335
- `dtype` (`astropy.nddata.NDData` attribute), 108
- `dtype` (`astropy.table.Row` attribute), 169
- `dtype` (`astropy.table.Table` attribute), 171
- `dump` (`astropy.io.fits.BinTableHDU` method), 635
- `dump` (`astropy.io.fits.CompImageHDU` method), 666
- `dump` (`astropy.units.Quantity` method), 70
- `dump` (`astropy.units.quantity.Quantity` method), 48
- `dumps` (`astropy.units.Quantity` method), 70
- `dumps` (`astropy.units.quantity.Quantity` method), 48
- `dumps` (`astropy.vo.client.vos_catalog.VOSBase` method), 892
- `DuplicateCatalogName`, 904
- `DuplicateCatalogURL`, 904
- `DynamicMatrixTransform` (class in `astropy.coordinates`), 278
- ## E
- `EarthLocation` (class in `astropy.coordinates`), 280
- `ecall` (`astropy.vo.samp.SAMPIntegratedClient` method), 935
- `ecall_all` (`astropy.vo.samp.SAMPIntegratedClient` method), 936
- `ecall_and_wait` (`astropy.vo.samp.SAMPIntegratedClient` method), 936
- `ediff1d` (`astropy.units.Quantity` method), 70
- `ediff1d` (`astropy.units.quantity.Quantity` method), 48
- `efunc` (`astropy.cosmology.FlatLambdaCDM` method), 839
- `efunc` (`astropy.cosmology.FlatwCDM` method), 842
- `efunc` (`astropy.cosmology.FLRW` method), 835
- `efunc` (`astropy.cosmology.LambdaCDM` method), 843
- `efunc` (`astropy.cosmology.wCDM` method), 851
- `element` (`astropy.utils.xml.writer.XMLWriter` method), 1012
- `ellipsoid` (`astropy.coordinates.EarthLocation` attribute), 281
- `EMConstant` (class in `astropy.constants`), 22
- `enable` (in module `astropy.units.cds`), 95
- `enable` (in module `astropy.units.imperial`), 92
- `enable_color` (`astropy.logger.AstropyLogger` method), 969
- `enable_exception_logging` (`astropy.logger.AstropyLogger` method), 969
- `enable_warnings_logging` (`astropy.logger.AstropyLogger` method), 969
- `end` (`astropy.utils.xml.writer.XMLWriter` method), 1012
- `end_line` (`astropy.io.ascii.BaseData` attribute), 706
- `enotify` (`astropy.vo.samp.SAMPIntegratedClient` method), 937
- `enotify_all` (`astropy.vo.samp.SAMPIntegratedClient` method), 937
- `entries` (`astropy.io.votable.tree.Group` attribute), 752
- `epoch` (`astropy.io.votable.tree.CooSys` attribute), 750
- `epoch_format` (`astropy.time.TimeCxcSec` attribute), 208
- `epoch_format` (`astropy.time.TimeGPS` attribute), 215
- `epoch_format` (`astropy.time.TimePlotDate` attribute), 219
- `epoch_format` (`astropy.time.TimeUnix` attribute), 221
- `epoch_prefix` (`astropy.time.TimeBesselianEpochString` attribute), 207
- `epoch_prefix` (`astropy.time.TimeJulianEpochString` attribute), 218
- `epoch_scale` (`astropy.time.TimeCxcSec` attribute), 208
- `epoch_scale` (`astropy.time.TimeGPS` attribute), 215
- `epoch_scale` (`astropy.time.TimePlotDate` attribute), 219
- `epoch_scale` (`astropy.time.TimeUnix` attribute), 221
- `epoch_to_jd` (`astropy.time.TimeBesselianEpoch` attribute), 207
- `epoch_to_jd` (`astropy.time.TimeBesselianEpochString` attribute), 207
- `epoch_to_jd` (`astropy.time.TimeJulianEpoch` attribute), 217
- `epoch_to_jd` (`astropy.time.TimeJulianEpochString` attribute), 218
- `epoch_val` (`astropy.time.TimeCxcSec` attribute), 208
- `epoch_val` (`astropy.time.TimeGPS` attribute), 215
- `epoch_val` (`astropy.time.TimePlotDate` attribute), 219
- `epoch_val` (`astropy.time.TimeUnix` attribute), 221
- `epoch_val2` (`astropy.time.TimeCxcSec` attribute), 208

- epoch_val2 (astropy.time.TimeGPS attribute), 215
- epoch_val2 (astropy.time.TimePlotDate attribute), 219
- epoch_val2 (astropy.time.TimeUnix attribute), 221
- eps (astropy.modeling.optimizers.Optimization attribute), 448
- eqcons (astropy.modeling.Model attribute), 395
- equinox (astropy.coordinates.AltAz attribute), 261
- equinox (astropy.coordinates.FK4 attribute), 284
- equinox (astropy.coordinates.FK4NoETerms attribute), 285
- equinox (astropy.coordinates.FK5 attribute), 286
- equinox (astropy.io.votable.tree.CooSys attribute), 750
- equinox (astropy.wcs.Wcsprm attribute), 351
- equivalencies (astropy.units.Quantity attribute), 67
- equivalencies (astropy.units.quantity.Quantity attribute), 45
- ereply() (astropy.vo.samp.SAMPIntegratedClient method), 938
- escapechar (astropy.io.ascii.DefaultSplitter attribute), 719
- eval() (astropy.modeling.Fittable1DModel method), 390
- eval() (astropy.modeling.Fittable2DModel method), 391
- eval() (astropy.modeling.functional_models.AiryDisk2D class method), 411
- eval() (astropy.modeling.functional_models.Beta1D static method), 413
- eval() (astropy.modeling.functional_models.Beta2D static method), 415
- eval() (astropy.modeling.functional_models.Box1D static method), 417
- eval() (astropy.modeling.functional_models.Box2D static method), 419
- eval() (astropy.modeling.functional_models.Const1D static method), 420
- eval() (astropy.modeling.functional_models.Const2D static method), 422
- eval() (astropy.modeling.functional_models.Disk2D static method), 423
- eval() (astropy.modeling.functional_models.Gaussian1D static method), 426
- eval() (astropy.modeling.functional_models.Gaussian2D static method), 428
- eval() (astropy.modeling.functional_models.GaussianAbsorption1D static method), 430
- eval() (astropy.modeling.functional_models.Linear1D static method), 432
- eval() (astropy.modeling.functional_models.Lorentz1D static method), 433
- eval() (astropy.modeling.functional_models.MexicanHat1D static method), 435
- eval() (astropy.modeling.functional_models.MexicanHat2D static method), 437
- eval() (astropy.modeling.functional_models.Redshift static method), 438
- eval() (astropy.modeling.functional_models.Ring2D static method), 439
- eval() (astropy.modeling.functional_models.Sine1D static method), 443
- eval() (astropy.modeling.functional_models.Trapezoid1D static method), 444
- eval() (astropy.modeling.functional_models.TrapezoidDisk2D static method), 447
- eval() (astropy.modeling.powerlaws.BrokenPowerLaw1D static method), 452
- eval() (astropy.modeling.powerlaws.ExponentialCutoffPowerLaw1D static method), 453
- eval() (astropy.modeling.powerlaws.LogParabola1D static method), 455
- eval() (astropy.modeling.powerlaws.PowerLaw1D static method), 456
- exception_logging_enabled() (astropy.logger.AstropyLogger method), 969
- exclude_names (astropy.io.ascii.BaseReader attribute), 710
- ExponentialCutoffPowerLaw1D (class in astropy.modeling.powerlaws), 452
- extend() (astropy.io.fits.CardList method), 632
- extend() (astropy.io.fits.Header method), 622
- extend() (astropy.utils.collections.HomogeneousList method), 985
- extra_attributes (astropy.io.votable.tree.Resource attribute), 756
- extrema (astropy.wcs.Tabprm attribute), 327
- ## F
- factors (astropy.modeling.functional_models.Scale attribute), 440
- Field (class in astropy.io.votable.tree), 745
- field() (astropy.io.fits.FITS_rec method), 651
- field() (astropy.io.fits.FITS_record method), 652
- field() (astropy.table.Table method), 175
- field_specifier (astropy.io.fits.Card attribute), 630
- FieldRef (class in astropy.io.votable.tree), 750
- fields (astropy.io.votable.tree.Table attribute), 754
- filebytes() (astropy.io.fits.BinTableHDU method), 636
- filebytes() (astropy.io.fits.CompImageHDU method), 667
- filebytes() (astropy.io.fits.GroupsHDU method), 612
- filebytes() (astropy.io.fits.ImageHDU method), 657
- filebytes() (astropy.io.fits.PrimaryHDU method), 606
- filebytes() (astropy.io.fits.TableHDU method), 644
- fileinfo() (astropy.io.fits.BinTableHDU method), 636
- fileinfo() (astropy.io.fits.CompImageHDU method), 667
- fileinfo() (astropy.io.fits.GroupsHDU method), 612
- fileinfo() (astropy.io.fits.HDUList method), 601
- fileinfo() (astropy.io.fits.ImageHDU method), 657
- fileinfo() (astropy.io.fits.PrimaryHDU method), 606
- fileinfo() (astropy.io.fits.TableHDU method), 644
- filename() (astropy.io.fits.HDUList method), 602

- fill() (astropy.units.Quantity method), 70
- fill() (astropy.units.quantity.Quantity method), 48
- fill_exclude_names (astropy.io.ascii.BaseData attribute), 706
- fill_include_names (astropy.io.ascii.BaseData attribute), 706
- fill_value (astropy.table.MaskedColumn attribute), 166
- filled() (astropy.table.MaskedColumn method), 167
- filled() (astropy.table.Table method), 175
- filter() (astropy.table.ColumnGroups method), 164
- filter() (astropy.table.TableGroups method), 187
- filter_list() (astropy.io.fits.CardList method), 632
- find_all_wcs() (in module astropy.wcs), 320
- find_api_page() (in module astropy.utils.misc), 979
- find_current_module() (in module astropy.utils.misc), 976
- find_equivalent_units() (astropy.units.UnitBase method), 79
- find_shortest_path() (astropy.coordinates.TransformGraph method), 304
- fit_deriv (astropy.modeling.FittableModel attribute), 392
- fit_deriv() (astropy.modeling.functional_models.Beta1D static method), 413
- fit_deriv() (astropy.modeling.functional_models.Beta2D static method), 415
- fit_deriv() (astropy.modeling.functional_models.Box1D class method), 417
- fit_deriv() (astropy.modeling.functional_models.Const1D static method), 420
- fit_deriv() (astropy.modeling.functional_models.Gaussian1D static method), 426
- fit_deriv() (astropy.modeling.functional_models.Gaussian2D static method), 428
- fit_deriv() (astropy.modeling.functional_models.GaussianAbsorption1D static method), 430
- fit_deriv() (astropy.modeling.functional_models.Linear1D static method), 432
- fit_deriv() (astropy.modeling.functional_models.Lorentz1D static method), 433
- fit_deriv() (astropy.modeling.functional_models.Redshift static method), 438
- fit_deriv() (astropy.modeling.functional_models.Sine1D static method), 443
- fit_deriv() (astropy.modeling.polynomial.Chebyshev1D method), 458
- fit_deriv() (astropy.modeling.polynomial.Chebyshev2D method), 460
- fit_deriv() (astropy.modeling.polynomial.Legendre1D method), 462
- fit_deriv() (astropy.modeling.polynomial.Legendre2D method), 464
- fit_deriv() (astropy.modeling.polynomial.Polynomial1D method), 466
- fit_deriv() (astropy.modeling.polynomial.Polynomial2D method), 467
- fit_deriv() (astropy.modeling.powerlaws.BrokenPowerLaw1D static method), 452
- fit_deriv() (astropy.modeling.powerlaws.ExponentialCutoffPowerLaw1D static method), 453
- fit_deriv() (astropy.modeling.powerlaws.LogParabola1D static method), 455
- fit_deriv() (astropy.modeling.powerlaws.PowerLaw1D static method), 456
- Fits (class in astropy.units.format), 86
- FITS_rec (class in astropy.io.fits), 651
- FITS_record (class in astropy.io.fits), 652
- FITSDiff (class in astropy.io.fits), 674
- FITSFixedWarning, 322
- fittable (astropy.modeling.FittableModel attribute), 392
- fittable (astropy.modeling.Model attribute), 395
- Fittable1DModel (class in astropy.modeling), 389
- Fittable2DModel (class in astropy.modeling), 390
- FittableModel (class in astropy.modeling), 391
- Fitter (class in astropy.modeling.fitting), 407
- fix() (astropy.wcs.WCS method), 335
- fix() (astropy.wcs.Wcsprm method), 356
- fix_id() (in module astropy.io.votable.xmlutil), 767
- fix_id() (in module astropy.utils.xml.check), 1008
- fixed (astropy.modeling.Model attribute), 395
- fixed (astropy.modeling.Parameter attribute), 398
- FixedWidth (class in astropy.io.ascii), 719
- FixedWidthData (class in astropy.io.ascii), 720
- FixedWidthHeader (class in astropy.io.ascii), 720
- FixedWidthNoHeader (class in astropy.io.ascii), 722
- FixedWidthSplitter (class in astropy.io.ascii), 722
- FixedWidthTwoLine (class in astropy.io.ascii), 723
- FK4 (class in astropy.coordinates), 283
- FK4NoEuler (class in astropy.coordinates), 284
- FK5 (class in astropy.coordinates), 285
- FlagCollection (class in astropy.nddata), 106
- flags (astropy.nddata.NDData attribute), 108
- flat (astropy.units.Quantity attribute), 67
- flat (astropy.units.quantity.Quantity attribute), 45
- FlatLambdaCDM (class in astropy.cosmology), 838
- Flatw0waCDM (class in astropy.cosmology), 840
- FlatwCDM (class in astropy.cosmology), 841
- FloatType (class in astropy.io.ascii), 724
- FLRW (class in astropy.cosmology), 827
- flush() (astropy.io.fits.HDUList method), 602
- flush() (astropy.utils.xml.writer.XMLWriter method), 1012
- fnpickle() (in module astropy.io.misc), 781
- fnunpickle() (in module astropy.io.misc), 782
- foc2pix() (astropy.wcs.Sip method), 325
- footprint_to_file() (astropy.wcs.WCS method), 335
- format (astropy.io.votable.tree.Table attribute), 754
- format (astropy.time.Time attribute), 203

- format() (astropy.coordinates.Angle method), 263
- format_exception() (in module astropy.utils.misc), 979
- format_input() (in module astropy.modeling), 389
- FORMATS (astropy.time.Time attribute), 203
- FORMATS (astropy.time.TimeDelta attribute), 210
- frame (astropy.coordinates.SkyCoord attribute), 297
- frame_set (astropy.coordinates.TransformGraph attribute), 304
- frame_specific_representation_info (astropy.coordinates.AltAz attribute), 261
- frame_specific_representation_info (astropy.coordinates.BaseCoordinateFrame attribute), 266
- frame_specific_representation_info (astropy.coordinates.FK4 attribute), 284
- frame_specific_representation_info (astropy.coordinates.FK4NoETerms attribute), 285
- frame_specific_representation_info (astropy.coordinates.FK5 attribute), 286
- frame_specific_representation_info (astropy.coordinates.Galactic attribute), 289
- frame_specific_representation_info (astropy.coordinates.GenericFrame attribute), 289
- frame_specific_representation_info (astropy.coordinates.ICRS attribute), 290
- FrameAttribute (class in astropy.coordinates), 286
- frequency (astropy.modeling.functional_models.Sine1D attribute), 443
- from_cartesian() (astropy.coordinates.BaseRepresentation method), 270
- from_cartesian() (astropy.coordinates.CartesianRepresentation class method), 272
- from_cartesian() (astropy.coordinates.CylindricalRepresentation class method), 276
- from_cartesian() (astropy.coordinates.PhysicsSphericalRepresentation class method), 294
- from_cartesian() (astropy.coordinates.SphericalRepresentation class method), 302
- from_cartesian() (astropy.coordinates.UnitSphericalRepresentation class method), 308
- from_columns() (astropy.io.fits.BinTableHDU class method), 637
- from_columns() (astropy.io.fits.CompImageHDU class method), 667
- from_columns() (astropy.io.fits.FITS_rec class method), 651
- from_columns() (astropy.io.fits.GroupsHDU class method), 612
- from_columns() (astropy.io.fits.TableHDU class method), 644
- from_geocentric() (astropy.coordinates.EarthLocation class method), 281
- from_geodetic() (astropy.coordinates.EarthLocation class method), 281
- from_json() (astropy.vo.client.vos_catalog.VOSDatabase class method), 895
- from_name() (astropy.coordinates.SkyCoord class method), 297
- from_registry() (astropy.vo.client.vos_catalog.VOSDatabase class method), 895
- from_representation() (astropy.coordinates.BaseRepresentation class method), 270
- from_table() (astropy.io.votable.tree.Table class method), 754
- from_table() (astropy.io.votable.tree.VOTableFile class method), 759
- from_table() (in module astropy.io.votable), 740
- from_table_column() (astropy.io.votable.tree.Field class method), 747
- from_table_column() (astropy.io.votable.tree.Link class method), 742
- from_table_column() (astropy.io.votable.tree.Values class method), 745
- fromdiff() (astropy.io.fits.FITSDiff class method), 675
- fromdiff() (astropy.io.fits.HDUDiff class method), 676
- fromdiff() (astropy.io.fits.HeaderDiff class method), 677
- fromdiff() (astropy.io.fits.ImageDataDiff class method), 679
- fromdiff() (astropy.io.fits.RawDataDiff class method), 679
- fromdiff() (astropy.io.fits.TableDataDiff class method), 681
- fromfile() (astropy.io.fits.HDUList class method), 602
- fromfile() (astropy.io.fits.Header class method), 623
- fromkeys() (astropy.io.fits.Header class method), 623
- fromstring() (astropy.io.fits.BinTableHDU class method), 637
- fromstring() (astropy.io.fits.Card class method), 630
- fromstring() (astropy.io.fits.CompImageHDU class method), 668
- fromstring() (astropy.io.fits.GroupsHDU class method), 613
- fromstring() (astropy.io.fits.HDUList class method), 602
- fromstring() (astropy.io.fits.Header class method), 623
- fromstring() (astropy.io.fits.ImageHDU class method), 658
- fromstring() (astropy.io.fits.PrimaryHDU class method), 606
- fromstring() (astropy.io.fits.TableHDU class method), 645
- fromtextfile() (astropy.io.fits.Header class method), 624
- fromTxtFile() (astropy.io.fits.Header method), 622
- FunctionTransform (class in astropy.coordinates), 287
- fwhm (astropy.modeling.functional_models.Lorentz1D attribute), 433

G

- Galactic (class in `astropy.coordinates`), 288
- `gamma` (`astropy.modeling.functional_models.Beta1D` attribute), 413
- `gamma` (`astropy.modeling.functional_models.Beta2D` attribute), 415
- `gamma` (`astropy.modeling.projections.Pix2Sky_AZP` attribute), 473
- `gamma` (`astropy.modeling.projections.Sky2Pix_AZP` attribute), 474
- `Gaussian1D` (class in `astropy.modeling.functional_models`), 423
- `Gaussian1DKernel` (class in `astropy.convolution`), 801
- `Gaussian2D` (class in `astropy.modeling.functional_models`), 426
- `Gaussian2DKernel` (class in `astropy.convolution`), 802
- `GaussianAbsorption1D` (class in `astropy.modeling.functional_models`), 429
- `Generic` (class in `astropy.units.format`), 84
- `GenericFrame` (class in `astropy.coordinates`), 289
- `geocentric` (`astropy.coordinates.EarthLocation` attribute), 281
- `geodetic` (`astropy.coordinates.EarthLocation` attribute), 281
- `get()` (`astropy.io.fits.Header` method), 624
- `get()` (`astropy.utils.state.ScienceState` class method), 997
- `get()` (`astropy.vo.client.async.AsyncBase` method), 903
- `get_axis_types()` (`astropy.wcs.WCS` method), 336
- `get_cache_dir()` (in module `astropy.config`), 953
- `get_catalog()` (`astropy.vo.client.vos_catalog.VOSDatabase` method), 896
- `get_catalog_by_url()` (`astropy.vo.client.vos_catalog.VOSDatabase` method), 896
- `get_catalogs()` (`astropy.vo.client.vos_catalog.VOSDatabase` method), 896
- `get_catalogs_by_url()` (`astropy.vo.client.vos_catalog.VOSDatabase` method), 896
- `get_cdelt()` (`astropy.wcs.Wcsprm` method), 356
- `get_col_type()` (`astropy.io.ascii.BaseHeader` method), 708
- `get_cols()` (`astropy.io.ascii.BaseHeader` method), 708
- `get_cols()` (`astropy.io.ascii.FixedWidthHeader` method), 721
- `get_comment()` (`astropy.io.fits.Header` method), 624
- `get_config()` (in module `astropy.config`), 954
- `get_config_dir()` (in module `astropy.config`), 954
- `get_converter()` (`astropy.units.UnitBase` method), 79
- `get_converter()` (`astropy.units.UnrecognizedUnit` method), 82
- `get_converter()` (in module `astropy.io.votable.converters`), 760
- `get_coosys_by_id()` (`astropy.io.votable.tree.VOTableFile` method), 759
- `get_cosmology_from_string()` (`astropy.cosmology.default_cosmology` static method), 845
- `get_current()` (in module `astropy.cosmology`), 823
- `get_current_unit_registry()` (in module `astropy.units`), 59
- `get_data_lines()` (`astropy.io.ascii.BaseData` method), 706
- `get_delta_ut1_utc()` (`astropy.time.Time` method), 204
- `get_field_by_id()` (`astropy.io.votable.tree.Table` method), 754
- `get_field_by_id()` (`astropy.io.votable.tree.VOTableFile` method), 759
- `get_field_by_id_or_name()` (`astropy.io.votable.tree.Table` method), 754
- `get_field_by_id_or_name()` (`astropy.io.votable.tree.VOTableFile` method), 759
- `get_fields_by_ctype()` (`astropy.io.votable.tree.Table` method), 754
- `get_fields_by_ctype()` (`astropy.io.votable.tree.VOTableFile` method), 759
- `get_file_contents()` (in module `astropy.utils.data`), 999
- `get_first_table()` (`astropy.io.votable.tree.VOTableFile` method), 759
- `get_fixedwidth_params()` (`astropy.io.ascii.FixedWidthHeader` method), 721
- `get_format()` (in module `astropy.units.format`), 83
- `get_format_name()` (`astropy.units.NamedUnit` method), 65
- `get_format_name()` (`astropy.units.UnrecognizedUnit` method), 82
- `get_formats()` (in module `astropy.io.registry`), 963
- `get_frame_attr_names()` (`astropy.coordinates.BaseCoordinateFrame` class method), 267
- `get_frame_attr_names()` (`astropy.coordinates.GenericFrame` method), 289
- `get_free_space_in_dir()` (in module `astropy.utils.data`), 1005
- `get_group_by_id()` (`astropy.io.votable.tree.Table` method), 754
- `get_group_by_id()` (`astropy.io.votable.tree.VOTableFile` method), 759
- `get_groups_by_ctype()` (`astropy.io.votable.tree.Table` method), 754
- `get_groups_by_ctype()` (`astropy.io.votable.tree.VOTableFile` method), 759
- `get_history()` (`astropy.io.fits.Header` method), 624
- `get_icrs_coordinates()` (in module `astropy.coordinates`), 257
- `get_include()` (in module `astropy.wcs`), 320

- [get_indentation\(\)](#) (astropy.utils.xml.writer.XMLWriter method), 1012
[get_indentation_spaces\(\)](#) (astropy.utils.xml.writer.XMLWriter method), 1012
[get_line\(\)](#) (astropy.io.ascii.FixedWidthHeader method), 722
[get_lines\(\)](#) (astropy.io.ascii.BaseInputter method), 709
[get_metadata\(\)](#) (astropy.vo.samp.SAMPHubProxy method), 926
[get_metadata\(\)](#) (astropy.vo.samp.SAMPIntegratedClient method), 938
[get_mtype_subtypes\(\)](#) (astropy.vo.samp.SAMPHubServer static method), 930
[get_name\(\)](#) (astropy.coordinates.BaseRepresentation class method), 270
[get_names\(\)](#) (astropy.coordinates.TransformGraph method), 305
[get_num_coeff\(\)](#) (astropy.modeling.polynomial.OrthoPolynomialBase method), 471
[get_num_coeff\(\)](#) (astropy.modeling.polynomial.PolynomialModel method), 472
[get_offset\(\)](#) (astropy.wcs.DistortionLookupTable method), 322
[get_pc\(\)](#) (astropy.wcs.Wcsprm method), 357
[get_physical_type\(\)](#) (in module astropy.units), 59
[get_pkg_data_contents\(\)](#) (in module astropy.utils.data), 1002
[get_pkg_data_filename\(\)](#) (in module astropy.utils.data), 1001
[get_pkg_data_filenames\(\)](#) (in module astropy.utils.data), 1004
[get_pkg_data_fileobj\(\)](#) (in module astropy.utils.data), 999
[get_pkg_data_fileobjs\(\)](#) (in module astropy.utils.data), 1003
[get_private_key\(\)](#) (astropy.vo.samp.SAMPClient method), 922
[get_private_key\(\)](#) (astropy.vo.samp.SAMPIntegratedClient method), 938
[get_ps\(\)](#) (astropy.wcs.Wcsprm method), 357
[get_public_id\(\)](#) (astropy.vo.samp.SAMPClient method), 922
[get_public_id\(\)](#) (astropy.vo.samp.SAMPIntegratedClient method), 939
[get_pv\(\)](#) (astropy.wcs.Wcsprm method), 357
[get_readable_fileobj\(\)](#) (in module astropy.utils.data), 998
[get_reader\(\)](#) (in module astropy.io.ascii), 701
[get_reader\(\)](#) (in module astropy.io.registry), 962
[get_ref\(\)](#) (astropy.io.votable.tree.FieldRef method), 751
[get_ref\(\)](#) (astropy.io.votable.tree.ParamRef method), 751
[get_registered_clients\(\)](#) (astropy.vo.samp.SAMPHubProxy method), 927
[get_registered_clients\(\)](#) (astropy.vo.samp.SAMPIntegratedClient method), 939
[get_remote_catalog_db\(\)](#) (in module astropy.vo.client.vos_catalog), 889
[get_str_vals\(\)](#) (astropy.io.ascii.BaseData method), 706
[get_subscribed_clients\(\)](#) (astropy.vo.samp.SAMPHubProxy method), 927
[get_subscribed_clients\(\)](#) (astropy.vo.samp.SAMPIntegratedClient method), 939
[get_subscriptions\(\)](#) (astropy.vo.samp.SAMPHubProxy method), 927
[get_subscriptions\(\)](#) (astropy.vo.samp.SAMPIntegratedClient method), 939
[get_table_by_id\(\)](#) (astropy.io.votable.tree.VOTableFile method), 759
[get_table_by_index\(\)](#) (astropy.io.votable.tree.VOTableFile method), 759
[get_tables_by_ctype\(\)](#) (astropy.io.votable.tree.VOTableFile method), 759
[get_tables_by_utype\(\)](#) (astropy.io.votable.tree.VOTableFile method), 759
[get_transform\(\)](#) (astropy.coordinates.TransformGraph method), 305
[get_type_map_key\(\)](#) (astropy.io.ascii.BaseHeader method), 708
[get_values_by_id\(\)](#) (astropy.io.votable.tree.VOTableFile method), 759
[get_writer\(\)](#) (in module astropy.io.ascii), 702
[get_writer\(\)](#) (in module astropy.io.registry), 962
[get_xml_encoding\(\)](#) (in module astropy.utils.xml.iterparser), 1009
[get_xml_iterator\(\)](#) (in module astropy.utils.xml.iterparser), 1009
[getdata\(\)](#) (in module astropy.io.fits), 597
[getheader\(\)](#) (in module astropy.io.fits), 598
[getter\(\)](#) (astropy.utils.misc.lazyproperty method), 981
[getval\(\)](#) (in module astropy.io.fits), 599
[Group](#) (class in astropy.io.fits), 618
[Group](#) (class in astropy.io.votable.tree), 751
[group_by\(\)](#) (astropy.table.Table method), 175
[GroupData](#) (class in astropy.io.fits), 617
[groups](#) (astropy.io.votable.tree.Table attribute), 754
[groups](#) (astropy.io.votable.tree.VOTableFile attribute), 758
[groups](#) (astropy.table.Table attribute), 171
[GroupsHDU](#) (class in astropy.io.fits), 610
- ## H
- [h](#) (astropy.cosmology.FLRW attribute), 830
[H\(\)](#) (astropy.cosmology.FLRW method), 830

- H() (in module `astropy.cosmology`), 821
- H0 (astropy.cosmology.FLRW attribute), 829
- `handle_queue()` (astropy.vo.samp.WebProfileDialog method), 942
- `has_cd()` (astropy.wcs.Wcsprm method), 357
- `has_cdi_ja()` (astropy.wcs.Wcsprm method), 358
- `has_crota()` (astropy.wcs.Wcsprm method), 358
- `has_crotaia()` (astropy.wcs.Wcsprm method), 358
- `has_data` (astropy.coordinates.BaseCoordinateFrame attribute), 266
- `has_massive_nu` (astropy.cosmology.FLRW attribute), 830
- `has_pc()` (astropy.wcs.Wcsprm method), 358
- `has_pci_ja()` (astropy.wcs.Wcsprm method), 358
- HDUdiff (class in `astropy.io.fits`), 676
- HDUList (class in `astropy.io.fits`), 600
- Header (class in `astropy.io.fits`), 619
- HeaderDiff (class in `astropy.io.fits`), 677
- `height` (astropy.coordinates.EarthLocation attribute), 281
- `hms` (astropy.coordinates.Angle attribute), 263
- HomogeneousList (class in `astropy.utils.collections`), 985
- `horner()` (astropy.modeling.polynomial.Polynomial1D method), 466
- `hour` (astropy.coordinates.Angle attribute), 263
- `href` (astropy.io.votable.tree.Link attribute), 742
- `hstack()` (in module `astropy.table`), 157
- HTML (class in `astropy.io.ascii`), 724
- `hubble_distance` (astropy.cosmology.FLRW attribute), 830
- `hubble_time` (astropy.cosmology.FLRW attribute), 830
- `human_file_size()` (in module `astropy.utils.console`), 987
- `human_time()` (in module `astropy.utils.console`), 986
- I
- ICRS (class in `astropy.coordinates`), 290
- ID (astropy.io.votable.tree.CooSys attribute), 750
- `id` (astropy.vo.samp.SAMPHubServer attribute), 929
- `identical` (astropy.io.fits.FITSDiff attribute), 675
- `identical` (astropy.io.fits.HDUdiff attribute), 676
- `identical` (astropy.io.fits.HeaderDiff attribute), 678
- `identical` (astropy.io.fits.ImageDataDiff attribute), 679
- `identical` (astropy.io.fits.RawDataDiff attribute), 680
- `identical` (astropy.io.fits.TableDataDiff attribute), 681
- `identify_format()` (in module `astropy.io.registry`), 962
- IllegalHourError, 290
- IllegalHourWarning, 291
- IllegalMinuteError, 291
- IllegalMinuteWarning, 291
- IllegalSecondError, 291
- IllegalSecondWarning, 292
- `image` (astropy.io.fits.Card attribute), 630
- ImageDataDiff (class in `astropy.io.fits`), 678
- ImageHDU (class in `astropy.io.fits`), 655
- `imgpix_matrix` (astropy.wcs.Wcsprm attribute), 351
- `imhorner()` (astropy.modeling.polynomial.OrthoPolynomialBase method), 471
- `in_subfmt` (astropy.time.Time attribute), 203
- `in_units()` (astropy.units.UnitBase method), 80
- `include_names` (astropy.io.ascii.BaseReader attribute), 710
- IncompatibleUncertaintiesException, 106
- `inconsistent_handler()` (astropy.io.ascii.BaseReader method), 710
- `inconsistent_handler()` (astropy.io.ascii.Csv method), 716
- InconsistentAxisTypesError, 323
- InconsistentTableError, 725
- `indent()` (in module `astropy.utils.misc`), 980
- `index` (astropy.table.Row attribute), 169
- `index()` (astropy.io.fits.CardList method), 632
- `index()` (astropy.io.fits.Header method), 624
- `index_columnn()` (astropy.table.Table method), 176
- `index_of()` (astropy.io.fits.CardList method), 632
- `index_of()` (astropy.io.fits.HDUList method), 603
- `indices` (astropy.table.ColumnGroups attribute), 164
- `indices` (astropy.table.TableGroups attribute), 187
- `ineqcons` (astropy.modeling.Model attribute), 395
- Info (class in `astropy.io.votable.tree`), 743
- `info()` (astropy.io.fits.ColDefs method), 650
- `info()` (astropy.io.fits.HDUList method), 603
- `info()` (in module `astropy.io.fits`), 595
- `infos` (astropy.io.votable.tree.Resource attribute), 756
- `infos` (astropy.io.votable.tree.Table attribute), 754
- `infos` (astropy.io.votable.tree.VOTableFile attribute), 758
- InheritDocstrings (class in `astropy.utils.misc`), 983
- InputParameterError, 392
- `insert()` (astropy.io.fits.CardList method), 633
- `insert()` (astropy.io.fits.HDUList method), 603
- `insert()` (astropy.io.fits.Header method), 625
- `insert()` (astropy.utils.collections.HomogeneousList method), 985
- `intercept` (astropy.modeling.functional_models.Linear1D attribute), 431
- IntType (class in `astropy.io.ascii`), 725
- `inv_efunc()` (astropy.cosmology.FlatLambdaCDM method), 839
- `inv_efunc()` (astropy.cosmology.FlatwCDM method), 842
- `inv_efunc()` (astropy.cosmology.FLRW method), 835
- `inv_efunc()` (astropy.cosmology.LambdaCDM method), 844
- `inv_efunc()` (astropy.cosmology.wCDM method), 851
- InvalidAccessURL, 904
- `invalidate_cache()` (astropy.coordinates.TransformGraph method), 305
- InvalidConfigurationItemWarning, 958
- InvalidCoordinateError, 323
- InvalidSubimageSpecificationError, 323
- InvalidTabularParametersError, 323
- InvalidTransformError, 323

- inverse() (astropy.modeling.functional_models.Redshift method), 438
- inverse() (astropy.modeling.functional_models.Scale method), 440
- inverse() (astropy.modeling.functional_models.Shift method), 441
- inverse() (astropy.modeling.Model method), 396
- inverse() (astropy.modeling.polynomial.SIP method), 469
- inverse() (astropy.modeling.projections.AffineTransformation2D method), 482
- inverse() (astropy.modeling.projections.Pix2Sky_AZP method), 473
- inverse() (astropy.modeling.projections.Pix2Sky_CAR method), 474
- inverse() (astropy.modeling.projections.Pix2Sky_CEA method), 476
- inverse() (astropy.modeling.projections.Pix2Sky_CYP method), 477
- inverse() (astropy.modeling.projections.Pix2Sky_MER method), 478
- inverse() (astropy.modeling.projections.Pix2Sky_SIN method), 479
- inverse() (astropy.modeling.projections.Pix2Sky_STG method), 480
- inverse() (astropy.modeling.projections.Pix2Sky_TAN method), 481
- inverse() (astropy.modeling.projections.Sky2Pix_AZP method), 474
- inverse() (astropy.modeling.projections.Sky2Pix_CAR method), 475
- inverse() (astropy.modeling.projections.Sky2Pix_CEA method), 476
- inverse() (astropy.modeling.projections.Sky2Pix_CYP method), 478
- inverse() (astropy.modeling.projections.Sky2Pix_MER method), 479
- inverse() (astropy.modeling.projections.Sky2Pix_SIN method), 479
- inverse() (astropy.modeling.projections.Sky2Pix_STG method), 480
- inverse() (astropy.modeling.projections.Sky2Pix_TAN method), 481
- inverse() (astropy.modeling.rotations.RotateCelestial2Native method), 484
- inverse() (astropy.modeling.rotations.RotateNative2Celestial method), 485
- inverse() (astropy.modeling.rotations.Rotation2D method), 486
- inverse() (astropy.modeling.SerialCompositeModel method), 400
- InverseSIP (class in astropy.modeling.polynomial), 460
- invert() (astropy.modeling.Model method), 396
- invlex_coeff() (astropy.modeling.polynomial.OrthoPolynomial method), 471
- invlex_coeff() (astropy.modeling.polynomial.Polynomial2D method), 468
- IOWarning (class in astropy.io.votable.exceptions), 779
- Ipac (class in astropy.io.ascii), 725
- IrreducibleUnit (class in astropy.units), 63
- is_blank (astropy.io.fits.Card attribute), 630
- is_bool (astropy.convolution.Kernel attribute), 804
- is_connected (astropy.vo.samp.SAMPHubProxy attribute), 925
- is_connected (astropy.vo.samp.SAMPIntegratedClient attribute), 932
- is_defaults() (astropy.io.votable.tree.Values method), 745
- is_empty() (astropy.io.votable.tree.Table method), 755
- is_equivalent() (astropy.units.UnitBase method), 80
- is_equivalent() (astropy.units.UnrecognizedUnit method), 82
- is_frame_attr_default() (astropy.coordinates.BaseCoordinateFrame method), 267
- is_path_hidden() (in module astropy.utils.misc), 980
- is_registered (astropy.vo.samp.SAMPClient attribute), 920
- is_running (astropy.vo.samp.SAMPClient attribute), 920
- is_running (astropy.vo.samp.SAMPHubServer attribute), 929
- is_scalar (astropy.time.Time attribute), 203
- is_transformable_to() (astropy.coordinates.BaseCoordinateFrame method), 267
- is_unity() (astropy.units.CompositeUnit method), 63
- is_unity() (astropy.units.Unit method), 77
- is_unity() (astropy.units.UnitBase method), 80
- is_unity() (astropy.units.UnrecognizedUnit method), 83
- is_unity() (astropy.wcs.Wcsprm method), 358
- is_votable() (in module astropy.io.votable), 740
- is_within_bounds() (astropy.coordinates.Angle method), 263
- isatty() (in module astropy.utils.console), 986
- isiterable() (in module astropy.utils.misc), 977
- isscalar (astropy.coordinates.BaseCoordinateFrame attribute), 266
- isscalar (astropy.coordinates.BaseRepresentation attribute), 270
- isscalar (astropy.units.Quantity attribute), 67
- isscalar (astropy.units.quantity.Quantity attribute), 45
- item() (astropy.units.Quantity method), 71
- item() (astropy.units.quantity.Quantity method), 49
- items() (astropy.io.fits.Header method), 625
- itemset() (astropy.units.Quantity method), 71
- itemset() (astropy.units.quantity.Quantity method), 49
- iter_coosys() (astropy.io.votable.tree.Resource method), 757
- iter_coosys() (astropy.io.votable.tree.VOTableFile method), 759

- [iter_fields_and_params\(\)](#) (astropy.io.votable.tree.Group method), 752
[iter_fields_and_params\(\)](#) (astropy.io.votable.tree.Resource method), 757
[iter_fields_and_params\(\)](#) (astropy.io.votable.tree.Table method), 755
[iter_fields_and_params\(\)](#) (astropy.io.votable.tree.VOTableFile method), 759
[iter_groups\(\)](#) (astropy.io.votable.tree.Group method), 752
[iter_groups\(\)](#) (astropy.io.votable.tree.Table method), 755
[iter_groups\(\)](#) (astropy.io.votable.tree.VOTableFile method), 759
[iter_tables\(\)](#) (astropy.io.votable.tree.Resource method), 757
[iter_tables\(\)](#) (astropy.io.votable.tree.VOTableFile method), 759
[iter_values\(\)](#) (astropy.io.votable.tree.VOTableFile method), 759
[iterate\(\)](#) (astropy.utils.console.ProgressBar class method), 989
[iteritems\(\)](#) (astropy.io.fits.Header method), 625
[iterkeys\(\)](#) (astropy.io.fits.Header method), 625
[itervalues\(\)](#) (astropy.io.fits.Header method), 625
- ## J
- [jd1](#) (astropy.time.Time attribute), 203
[jd2](#) (astropy.time.Time attribute), 203
[jd_to_epoch](#) (astropy.time.TimeBesselianEpoch attribute), 207
[jd_to_epoch](#) (astropy.time.TimeBesselianEpochString attribute), 207
[jd_to_epoch](#) (astropy.time.TimeJulianEpoch attribute), 217
[jd_to_epoch](#) (astropy.time.TimeJulianEpochString attribute), 218
[join\(\)](#) (astropy.io.ascii.BaseSplitter method), 712
[join\(\)](#) (astropy.io.ascii.DefaultSplitter method), 719
[join\(\)](#) (astropy.io.ascii.FixedWidthSplitter method), 723
[join\(\)](#) (in module astropy.table), 158
[JointFitter](#) (class in astropy.modeling.fitting), 406
[JsonCustomEncoder](#) (class in astropy.utils.misc), 982
- ## K
- [K](#) (astropy.wcs.Tabprm attribute), 327
[keep_columns\(\)](#) (astropy.table.Table method), 176
[Kernel](#) (class in astropy.convolution), 803
[Kernel1D](#) (class in astropy.convolution), 805
[Kernel2D](#) (class in astropy.convolution), 805
[kernel_arithmetics\(\)](#) (in module astropy.convolution), 797
[key](#) (astropy.io.fits.Card attribute), 630
[key_colnames](#) (astropy.table.TableGroups attribute), 187
[keys](#) (astropy.table.ColumnGroups attribute), 164
[keys](#) (astropy.table.TableGroups attribute), 187
[keys\(\)](#) (astropy.io.fits.CardList method), 633
[keys\(\)](#) (astropy.io.fits.Header method), 625
[keys\(\)](#) (astropy.table.Table method), 177
[keys\(\)](#) (astropy.table.TableColumns method), 186
[keyword](#) (astropy.io.fits.Card attribute), 630
[kpc_comoving_per_arcmin\(\)](#) (astropy.cosmology.FLRW method), 836
[kpc_comoving_per_arcmin\(\)](#) (in module astropy.cosmology), 823
[kpc_proper_per_arcmin\(\)](#) (astropy.cosmology.FLRW method), 836
[kpc_proper_per_arcmin\(\)](#) (in module astropy.cosmology), 824
- ## L
- [LabeledInput](#) (class in astropy.modeling), 392
[lam](#) (astropy.modeling.projections.Pix2Sky_CEA attribute), 475
[lam](#) (astropy.modeling.projections.Pix2Sky_CYP attribute), 477
[lam](#) (astropy.modeling.projections.Sky2Pix_CEA attribute), 476
[lam](#) (astropy.modeling.projections.Sky2Pix_CYP attribute), 478
[LambdaCDM](#) (class in astropy.cosmology), 842
[lat](#) (astropy.coordinates.SphericalRepresentation attribute), 301
[lat](#) (astropy.coordinates.UnitSphericalRepresentation attribute), 308
[lat](#) (astropy.time.Time attribute), 204
[lat](#) (astropy.wcs.Wcsprm attribute), 351
[Latex](#) (class in astropy.io.ascii), 727
[Latex](#) (class in astropy.units.format), 86
[latitude](#) (astropy.coordinates.EarthLocation attribute), 281
[Latitude](#) (class in astropy.coordinates), 292
[latpole](#) (astropy.wcs.Wcsprm attribute), 351
[lattyp](#) (astropy.wcs.Wcsprm attribute), 351
[lazyproperty](#) (class in astropy.utils.misc), 980
[leastsquare\(\)](#) (in module astropy.modeling.statistic), 483
[Legendre1D](#) (class in astropy.modeling.polynomial), 461
[Legendre2D](#) (class in astropy.modeling.polynomial), 463
[length](#) (astropy.io.fits.Card attribute), 630
[LevMarLSQFitter](#) (class in astropy.modeling.fitting), 402
[linear](#) (astropy.modeling.FittableModel attribute), 392
[linear](#) (astropy.modeling.functional_models.Const1D attribute), 420
[linear](#) (astropy.modeling.functional_models.Const2D attribute), 421
[linear](#) (astropy.modeling.functional_models.Linear1D attribute), 431
[linear](#) (astropy.modeling.functional_models.Scale attribute), 440
[linear](#) (astropy.modeling.Model attribute), 395

- Linear1D (class in `astropy.modeling.functional_models`), 430
- LinearLSQFitter (class in `astropy.modeling.fitting`), 401
- Link (class in `astropy.io.votable.tree`), 742
- links (`astropy.io.votable.tree.Field` attribute), 746
- links (`astropy.io.votable.tree.Resource` attribute), 756
- links (`astropy.io.votable.tree.Table` attribute), 754
- list() (`astropy.units.Quantity` method), 72
- list() (`astropy.units.quantity.Quantity` method), 50
- list_catalogs() (`astropy.vo.client.vos_catalog.VOSDatabase` method), 896
- list_catalogs() (in module `astropy.vo.client.conesearch`), 899
- list_catalogs() (in module `astropy.vo.client.vos_catalog`), 891
- list_catalogs_by_url() (`astropy.vo.client.vos_catalog.VOSDatabase` method), 897
- list_cats() (`astropy.vo.validator.inspect.ConeSearchResults` method), 907
- lng (`astropy.wcs.Wcsprm` attribute), 351
- lngtyp (`astropy.wcs.Wcsprm` attribute), 352
- load() (`astropy.io.fits.BinTableHDU` class method), 638
- load() (`astropy.io.fits.CompImageHDU` class method), 668
- location (`astropy.coordinates.AltAz` attribute), 261
- log_exceptions (`astropy.logger.Conf` attribute), 968
- log_file_format (`astropy.logger.Conf` attribute), 968
- log_file_level (`astropy.logger.Conf` attribute), 968
- log_file_path (`astropy.logger.Conf` attribute), 968
- log_level (`astropy.logger.Conf` attribute), 968
- log_to_file (`astropy.logger.Conf` attribute), 968
- log_to_file() (`astropy.logger.AstropyLogger` method), 969
- log_to_list() (`astropy.logger.AstropyLogger` method), 970
- log_warnings (`astropy.logger.Conf` attribute), 968
- logarithmic() (in module `astropy.units`), 59
- logarithmic() (in module `astropy.units.equivalencies`), 99
- LoggingError, 971
- LogParabola1D (class in `astropy.modeling.powerlaws`), 453
- lon (`astropy.coordinates.SphericalRepresentation` attribute), 301
- lon (`astropy.coordinates.UnitSphericalRepresentation` attribute), 308
- lon (`astropy.time.Time` attribute), 204
- long_names (`astropy.units.NamedUnit` attribute), 65
- longitude (`astropy.coordinates.EarthLocation` attribute), 281
- Longitude (class in `astropy.coordinates`), 292
- lonpole (`astropy.wcs.Wcsprm` attribute), 352
- lookback_time() (`astropy.cosmology.FLRW` method), 836
- lookback_time() (in module `astropy.cosmology`), 824
- lookup_name() (`astropy.coordinates.TransformGraph` method), 305
- Lorentz1D (class in `astropy.modeling.functional_models`), 432
- luminosity_distance() (`astropy.cosmology.FLRW` method), 836
- luminosity_distance() (in module `astropy.cosmology`), 824
- ## M
- M (`astropy.wcs.Tabprm` attribute), 327
- m_nu (`astropy.cosmology.FLRW` attribute), 830
- make_validation_report() (in module `astropy.io.votable.validator`), 766
- makeRecord() (`astropy.logger.AstropyLogger` method), 970
- map (`astropy.wcs.Tabprm` attribute), 327
- map() (`astropy.utils.console.ProgressBar` class method), 989
- mask (`astropy.nddata.NDData` attribute), 108
- mask (`astropy.table.Table` attribute), 171
- masked (`astropy.table.Table` attribute), 171
- MaskedColumn (class in `astropy.table`), 165
- masks() (`astropy.io.ascii.BaseData` method), 706
- mass_energy() (in module `astropy.units`), 59
- mass_energy() (in module `astropy.units.equivalencies`), 98
- match_coordinates_3d() (in module `astropy.coordinates`), 257
- match_coordinates_sky() (in module `astropy.coordinates`), 258
- match_to_catalog_3d() (`astropy.coordinates.SkyCoord` method), 297
- match_to_catalog_sky() (`astropy.coordinates.SkyCoord` method), 298
- matrix (`astropy.modeling.projections.AffineTransformation2D` attribute), 482
- max (`astropy.io.votable.tree.Values` attribute), 744
- max (`astropy.modeling.Parameter` attribute), 398
- max() (`astropy.units.Quantity` method), 72
- max() (`astropy.units.quantity.Quantity` method), 50
- max_inclusive (`astropy.io.votable.tree.Values` attribute), 744
- maxiter (`astropy.modeling.optimizers.Optimization` attribute), 448
- mean (`astropy.modeling.functional_models.Gaussian1D` attribute), 425
- mean (`astropy.modeling.functional_models.GaussianAbsorption1D` attribute), 430
- mean() (`astropy.units.Quantity` method), 72
- mean() (`astropy.units.quantity.Quantity` method), 50
- median_absolute_deviation() (in module `astropy.stats`), 862

- merge() (astropy.vo.client.vos_catalog.VOSDatabase method), 897
- meta (astropy.table.Row attribute), 170
- MexicanHat1D (class in astropy.modeling.functional_models), 434
- MexicanHat1DKernel (class in astropy.convolution), 806
- MexicanHat2D (class in astropy.modeling.functional_models), 435
- MexicanHat2DKernel (class in astropy.convolution), 807
- mhorner() (astropy.modeling.polynomial.Polynomial2D method), 468
- min (astropy.io.votable.tree.Values attribute), 744
- min (astropy.modeling.Parameter attribute), 398
- min() (astropy.units.Quantity method), 73
- min() (astropy.units.quantity.Quantity method), 51
- min_inclusive (astropy.io.votable.tree.Values attribute), 744
- MissingCatalog, 904
- MissingDataAssociationException, 106
- mix() (astropy.wcs.Wcsprm method), 358
- mjdavg (astropy.wcs.Wcsprm attribute), 352
- mjdobs (astropy.wcs.Wcsprm attribute), 352
- model (astropy.convolution.Kernel attribute), 804
- Model (class in astropy.modeling), 393
- Model1DKernel (class in astropy.convolution), 808
- Model2DKernel (class in astropy.convolution), 809
- model_constraints (astropy.modeling.Model attribute), 395
- model_set_axis (astropy.modeling.Model attribute), 395
- ModelDefinitionError, 397
- more() (astropy.table.Column method), 162
- more() (astropy.table.MaskedColumn method), 167
- more() (astropy.table.Table method), 177
- mu (astropy.modeling.projections.Pix2Sky_AZP attribute), 473
- mu (astropy.modeling.projections.Pix2Sky_CYP attribute), 477
- mu (astropy.modeling.projections.Sky2Pix_AZP attribute), 474
- mu (astropy.modeling.projections.Sky2Pix_CYP attribute), 478
- multiply() (astropy.nddata.NDData method), 110
- ## N
- n_inputs (astropy.modeling.Fittable2DModel attribute), 391
- n_inputs (astropy.modeling.Model attribute), 395
- n_inputs (astropy.modeling.polynomial.InverseSIP attribute), 461
- n_inputs (astropy.modeling.polynomial.OrthoPolynomialBase attribute), 470
- n_inputs (astropy.modeling.polynomial.PolynomialModel attribute), 471
- n_inputs (astropy.modeling.polynomial.SIP attribute), 469
- n_inputs (astropy.modeling.projections.AffineTransformation2D attribute), 482
- n_inputs (astropy.modeling.rotations.Rotation2D attribute), 485
- n_outputs (astropy.modeling.Fittable2DModel attribute), 391
- n_outputs (astropy.modeling.Model attribute), 395
- n_outputs (astropy.modeling.polynomial.InverseSIP attribute), 461
- n_outputs (astropy.modeling.polynomial.OrthoPolynomialBase attribute), 470
- n_outputs (astropy.modeling.polynomial.PolynomialModel attribute), 471
- n_outputs (astropy.modeling.polynomial.SIP attribute), 469
- n_outputs (astropy.modeling.projections.AffineTransformation2D attribute), 482
- n_outputs (astropy.modeling.rotations.Rotation2D attribute), 485
- name (astropy.constants.Constant attribute), 22
- name (astropy.coordinates.AltAz attribute), 261
- name (astropy.coordinates.BaseCoordinateFrame attribute), 266
- name (astropy.coordinates.FK4 attribute), 284
- name (astropy.coordinates.FK4NoETerms attribute), 285
- name (astropy.coordinates.FK5 attribute), 286
- name (astropy.coordinates.Galactic attribute), 289
- name (astropy.coordinates.GenericFrame attribute), 289
- name (astropy.coordinates.ICRS attribute), 290
- name (astropy.io.votable.tree.Info attribute), 743
- name (astropy.modeling.Parameter attribute), 398
- name (astropy.table.Column attribute), 161
- name (astropy.table.MaskedColumn attribute), 166
- name (astropy.time.TimeBesselianEpoch attribute), 207
- name (astropy.time.TimeBesselianEpochString attribute), 207
- name (astropy.time.TimeCxcSec attribute), 208
- name (astropy.time.TimeDatetime attribute), 209
- name (astropy.time.TimeDeltaJD attribute), 211
- name (astropy.time.TimeDeltaSec attribute), 211
- name (astropy.time.TimeGPS attribute), 215
- name (astropy.time.TimeISO attribute), 215
- name (astropy.time.TimeISOT attribute), 216
- name (astropy.time.TimeJD attribute), 217
- name (astropy.time.TimeJulianEpoch attribute), 217
- name (astropy.time.TimeJulianEpochString attribute), 218
- name (astropy.time.TimeMJD attribute), 218
- name (astropy.time.TimePlotDate attribute), 219
- name (astropy.time.TimeUnix attribute), 221
- name (astropy.time.TimeYearDayTime attribute), 221
- name (astropy.units.format.Fits attribute), 86

- name (astropy.units.NamedUnit attribute), 65
 name (astropy.units.UnitBase attribute), 78
 name (astropy.wcs.Wcsprm attribute), 352
 NamedUnit (class in astropy.units), 64
 names (astropy.io.ascii.BaseHeader attribute), 708
 names (astropy.io.ascii.BaseReader attribute), 710
 names (astropy.io.fits.FITS_rec attribute), 652
 names (astropy.units.NamedUnit attribute), 65
 names (astropy.units.UnitBase attribute), 78
 nansum() (astropy.units.Quantity method), 73
 nansum() (astropy.units.quantity.Quantity method), 51
 naxis (astropy.wcs.Wcsprm attribute), 352
 nc (astropy.wcs.Tabprm attribute), 328
 NDData (class in astropy.nddata), 106
 ndim (astropy.nddata.NDData attribute), 108
 NDUncertainty (class in astropy.nddata), 111
 Neff (astropy.cosmology.FLRW attribute), 829
 new_table() (in module astropy.io.fits), 652
 next() (astropy.table.Table method), 178
 no_continue (astropy.io.ascii.ContinuationLinesInputter attribute), 715
 NoHeader (class in astropy.io.ascii), 729
 noncritical_warnings (astropy.vo.validator.Conf attribute), 905
 NonseparableSubimageCoordinateSystemError, 323
 normalization (astropy.convolution.Kernel attribute), 804
 normalize() (astropy.convolution.Kernel method), 804
 normalize_keyword() (astropy.io.fits.Card class method), 630
 NoSolutionError, 323
 notify() (astropy.vo.samp.SAMPHubProxy method), 927
 notify() (astropy.vo.samp.SAMPIntegratedClient method), 939
 notify_all() (astropy.vo.samp.SAMPHubProxy method), 927
 notify_all() (astropy.vo.samp.SAMPIntegratedClient method), 939
 NoType (class in astropy.io.ascii), 729
 now() (astropy.time.Time class method), 205
 NoWcsKeywordsFoundError, 323
 nrows (astropy.io.votable.tree.Table attribute), 754
 nu_relative_density() (astropy.cosmology.FLRW method), 837
 null (astropy.io.votable.tree.Values attribute), 744
 NumpyRNGContext (class in astropy.utils.misc), 981
 NumType (class in astropy.io.ascii), 729
- O**
- object_attrs() (astropy.utils.xml.writer.XMLWriter static method), 1012
 objective_function() (astropy.modeling.fitting.Fitter method), 407
 objective_function() (astropy.modeling.fitting.JointFitter method), 407
 objective_function() (astropy.modeling.fitting.LevMarLSQFitter method), 403
 obsgeo (astropy.wcs.Wcsprm attribute), 352
 obstime (astropy.coordinates.AltAz attribute), 261
 obstime (astropy.coordinates.FK4 attribute), 284
 obstime (astropy.coordinates.FK4NoETerms attribute), 285
 Ode() (astropy.cosmology.FLRW method), 830
 Ode0 (astropy.cosmology.FLRW attribute), 829
 offsets (astropy.modeling.functional_models.Shift attribute), 441
 Ogamma() (astropy.cosmology.FLRW method), 830
 Ogamma0 (astropy.cosmology.FLRW attribute), 829
 OGIP (class in astropy.units.format), 87
 Ok() (astropy.cosmology.FLRW method), 831
 Ok0 (astropy.cosmology.FLRW attribute), 829
 Om() (astropy.cosmology.FLRW method), 831
 Om0 (astropy.cosmology.FLRW attribute), 829
 Onu() (astropy.cosmology.FLRW method), 831
 Onu0 (astropy.cosmology.FLRW attribute), 829
 open() (in module astropy.io.fits), 594
 OperandTypeError, 202
 opt_method (astropy.modeling.optimizers.Optimization attribute), 448
 Optimization (class in astropy.modeling.optimizers), 448
 options (astropy.io.votable.tree.Values attribute), 744
 OrthoPolynomialBase (class in astropy.modeling.polynomial), 469
 out_subfmt (astropy.time.Time attribute), 204
 output() (astropy.io.votable.converters.Converter method), 762
- P**
- p0 (astropy.wcs.Tabprm attribute), 328
 p2s() (astropy.wcs.Wcsprm method), 360
 p4_pix2foc() (astropy.wcs.WCS method), 336
 par() (astropy.io.fits.Group method), 618
 par() (astropy.io.fits.GroupData method), 617
 parallax() (in module astropy.units), 59
 parallax() (in module astropy.units.equivalencies), 95
 Param (class in astropy.io.votable.tree), 748
 param_dim (astropy.modeling.Model attribute), 396
 param_names (astropy.modeling.functional_models.AiryDisk2D attribute), 411
 param_names (astropy.modeling.functional_models.Beta1D attribute), 413
 param_names (astropy.modeling.functional_models.Beta2D attribute), 415
 param_names (astropy.modeling.functional_models.Box1D attribute), 416
 param_names (astropy.modeling.functional_models.Box2D attribute), 418

- param_names (astropy.modeling.functional_models.Const1D attribute), 420
- param_names (astropy.modeling.functional_models.Const2D attribute), 421
- param_names (astropy.modeling.functional_models.Disk2D attribute), 423
- param_names (astropy.modeling.functional_models.Gaussian1D attribute), 425
- param_names (astropy.modeling.functional_models.Gaussian2D attribute), 428
- param_names (astropy.modeling.functional_models.GaussianAbsorption1D attribute), 430
- param_names (astropy.modeling.functional_models.Linear1D attribute), 432
- param_names (astropy.modeling.functional_models.Lorentz1D attribute), 433
- param_names (astropy.modeling.functional_models.MexicanHat1D attribute), 435
- param_names (astropy.modeling.functional_models.MexicanHat2D attribute), 437
- param_names (astropy.modeling.functional_models.Redshift attribute), 438
- param_names (astropy.modeling.functional_models.Ring2D attribute), 439
- param_names (astropy.modeling.functional_models.Scale attribute), 440
- param_names (astropy.modeling.functional_models.Shift attribute), 441
- param_names (astropy.modeling.functional_models.Sine1D attribute), 443
- param_names (astropy.modeling.functional_models.Trapezoid1D attribute), 444
- param_names (astropy.modeling.functional_models.Trapezoid2D attribute), 447
- param_names (astropy.modeling.Model attribute), 396
- param_names (astropy.modeling.powerlaws.BrokenPowerLaw attribute), 452
- param_names (astropy.modeling.powerlaws.ExponentialCutoffPowerLaw1D attribute), 453
- param_names (astropy.modeling.powerlaws.LogParabola1D attribute), 454
- param_names (astropy.modeling.powerlaws.PowerLaw1D attribute), 456
- param_names (astropy.modeling.projections.AffineTransformation2D attribute), 482
- param_names (astropy.modeling.projections.Pix2Sky_AZP attribute), 473
- param_names (astropy.modeling.projections.Pix2Sky_CEA attribute), 475
- param_names (astropy.modeling.projections.Pix2Sky_CYP attribute), 477
- param_names (astropy.modeling.projections.Sky2Pix_AZP attribute), 474
- param_names (astropy.modeling.projections.Sky2Pix_CEA attribute), 476
- param_names (astropy.modeling.projections.Sky2Pix_CYP attribute), 478
- param_names (astropy.modeling.rotations.Rotation2D attribute), 485
- param_sets (astropy.modeling.Model attribute), 396
- Parameter (class in astropy.modeling), 397
- parameter_constraints (astropy.modeling.Model attribute), 396
- AbstractBase1D (astropy.modeling.Model attribute), 396
- ParamRef (class in astropy.io.votable.tree), 751
- params (astropy.io.votable.tree.Resource attribute), 756
- params (astropy.io.votable.tree.Table attribute), 754
- params (astropy.io.votable.tree.VOTableFile attribute), 758
- params (astropy.vo.samp.SAMPHubServer attribute), 929
- parent_nddata (astropy.nddata.NDUncertainty attribute), 111
- parent_nddata (astropy.nddata.StdDevUncertainty attribute), 113
- parnames (astropy.io.fits.GroupsHDU attribute), 613
- parse() (astropy.io.votable.converters.Converter method), 763
- parse() (astropy.io.votable.tree.Field method), 747
- parse() (astropy.io.votable.tree.Group method), 752
- parse() (astropy.io.votable.tree.Resource method), 757
- parse() (astropy.io.votable.tree.Table method), 755
- parse() (astropy.io.votable.tree.Values method), 745
- parse() (astropy.io.votable.tree.VOTableFile method), 759
- parse() (astropy.units.format.Base method), 84
- parse() (astropy.units.format.CDS method), 85
- parse() (astropy.units.format.Generic method), 85
- parse() (astropy.units.format.OGIP method), 87
- parse() (astropy.units.format.VOUnit method), 88
- parse() (in module astropy.io.votable), 738
- parse_scalar() (astropy.io.votable.converters.Converter method), 763
- parse_single_table() (in module astropy.io.votable), 739
- parse_ucd() (in module astropy.io.votable.ucd), 763
- parse_vowarning() (in module astropy.io.votable.exceptions), 779
- pc (astropy.wcs.Wcsprm attribute), 352
- padding2D (astropy.io.votable.Conf attribute), 741
- pformat() (astropy.table.Column method), 162
- pformat() (astropy.table.MaskedColumn method), 168
- pformat() (astropy.table.Table method), 178
- phi (astropy.coordinates.CylindricalRepresentation attribute), 275
- phi (astropy.coordinates.PhysicsSphericalRepresentation attribute), 294
- phi0 (astropy.wcs.Wcsprm attribute), 353
- physical_type (astropy.units.UnitBase attribute), 78

- PhysicsSphericalRepresentation (class in astropy.coordinates), 293
- ping() (astropy.vo.samp.SAMPHubProxy method), 927
- ping() (astropy.vo.samp.SAMPIntegratedClient method), 939
- pix2foc() (astropy.wcs.Sip method), 326
- pix2foc() (astropy.wcs.WCS method), 337
- Pix2Sky_AZP (class in astropy.modeling.projections), 472
- Pix2Sky_CAR (class in astropy.modeling.projections), 474
- Pix2Sky_CEA (class in astropy.modeling.projections), 475
- Pix2Sky_CYP (class in astropy.modeling.projections), 476
- Pix2Sky_MER (class in astropy.modeling.projections), 478
- Pix2Sky_SIN (class in astropy.modeling.projections), 479
- Pix2Sky_STG (class in astropy.modeling.projections), 480
- Pix2Sky_TAN (class in astropy.modeling.projections), 480
- piximg_matrix (astropy.wcs.Wcsprm attribute), 353
- plot() (astropy.utils.timer.RunTimePredictor method), 995
- Polynomial1D (class in astropy.modeling.polynomial), 464
- Polynomial2D (class in astropy.modeling.polynomial), 466
- PolynomialModel (class in astropy.modeling.polynomial), 471
- pop() (astropy.io.fits.CardList method), 633
- pop() (astropy.io.fits.Header method), 625
- popitem() (astropy.io.fits.Header method), 625
- position_angle() (astropy.coordinates.SkyCoord method), 299
- position_line (astropy.io.ascii.FixedWidthHeader attribute), 721
- PowerLaw1D (class in astropy.modeling.powerlaws), 455
- powers (astropy.units.CompositeUnit attribute), 62
- powers (astropy.units.UnitBase attribute), 78
- pprint() (astropy.table.Column method), 163
- pprint() (astropy.table.MaskedColumn method), 168
- pprint() (astropy.table.Table method), 178
- precision (astropy.io.votable.tree.Field attribute), 746
- precision (astropy.time.Time attribute), 204
- predict_search() (in module astropy.vo.client.conesearch), 900
- predict_time() (astropy.utils.timer.RunTimePredictor method), 996
- PrefixUnit (class in astropy.units), 65
- PrimaryHDU (class in astropy.io.fits), 604
- print_cat() (astropy.vo.validator.inspect.ConeSearchResults method), 907
- print_code_line() (in module astropy.utils.console), 987
- print_contents() (astropy.wcs.Tabprm method), 328
- print_contents() (astropy.wcs.Wcsprm method), 361
- printwcs() (astropy.wcs.WCS method), 337
- process_line() (astropy.io.ascii.BaseSplitter method), 712
- process_line() (astropy.io.ascii.DefaultSplitter method), 719
- process_line() (astropy.io.ascii.WhitespaceSplitter method), 731
- process_lines() (astropy.io.ascii.BaseData method), 706
- process_lines() (astropy.io.ascii.BaseHeader method), 708
- process_lines() (astropy.io.ascii.BaseInputter method), 709
- process_lines() (astropy.io.ascii.ContinuationLinesInputter method), 716
- process_val() (astropy.io.ascii.BaseSplitter method), 712
- prod() (astropy.units.Quantity method), 73
- prod() (astropy.units.quantity.Quantity method), 51
- ProgressBar (class in astropy.utils.console), 988
- ProgressBarOrSpinner (class in astropy.utils.console), 990
- propagate_add() (astropy.nddata.NDUncertainty method), 112
- propagate_add() (astropy.nddata.StdDevUncertainty method), 114
- propagate_divide() (astropy.nddata.NDUncertainty method), 112
- propagate_divide() (astropy.nddata.StdDevUncertainty method), 114
- propagate_multiply() (astropy.nddata.NDUncertainty method), 112
- propagate_multiply() (astropy.nddata.StdDevUncertainty method), 114
- propagate_subtract() (astropy.nddata.NDUncertainty method), 112
- propagate_subtract() (astropy.nddata.StdDevUncertainty method), 115
- ptp() (astropy.units.Quantity method), 73
- ptp() (astropy.units.quantity.Quantity method), 51
- put() (astropy.units.Quantity method), 73
- put() (astropy.units.quantity.Quantity method), 51
- ## Q
- Quantity (class in astropy.units), 65
- Quantity (class in astropy.units.quantity), 43
- quotechar (astropy.io.ascii.DefaultSplitter attribute), 719
- quoting (astropy.io.ascii.DefaultSplitter attribute), 719
- ## R
- r (astropy.coordinates.PhysicsSphericalRepresentation attribute), 294
- R_0 (astropy.modeling.functional_models.Disk2D attribute), 423

- R_0 (astropy.modeling.functional_models.TrapezoidDisk2D recommended_units attribute), 447
- r_in (astropy.modeling.functional_models.Ring2D attribute), 439
- radesys (astropy.wcs.Wcsprm attribute), 353
- radius (astropy.modeling.functional_models.AiryDisk2D attribute), 411
- RangeError, 295
- RawDataDiff (class in astropy.io.fits), 679
- rawkeyword (astropy.io.fits.Card attribute), 631
- rawvalue (astropy.io.fits.Card attribute), 631
- Rdb (class in astropy.io.ascii), 729
- read() (astropy.io.ascii.BaseReader method), 711
- read() (astropy.io.ascii.Cds method), 714
- read() (astropy.io.ascii.HTML method), 725
- read() (astropy.io.ascii.SExtractor method), 730
- read() (astropy.nddata.NDData class method), 110
- read() (astropy.table.Table class method), 179
- read() (in module astropy.io.ascii), 703
- read() (in module astropy.io.registry), 963
- read_table_hdf5() (in module astropy.io.misc.hdf5), 782
- readall() (astropy.io.fits.HDUList method), 603
- readfrom() (astropy.io.fits.BinTableHDU class method), 639
- readfrom() (astropy.io.fits.CompImageHDU class method), 670
- readfrom() (astropy.io.fits.GroupsHDU class method), 613
- readfrom() (astropy.io.fits.ImageHDU class method), 658
- readfrom() (astropy.io.fits.PrimaryHDU class method), 607
- readfrom() (astropy.io.fits.TableHDU class method), 645
- realize_frame() (astropy.coordinates.BaseCoordinateFrame method), 268
- receive_call() (astropy.vo.samp.SAMPClient method), 922
- receive_call() (astropy.vo.samp.SAMPIntegratedClient method), 939
- receive_notification() (astropy.vo.samp.SAMPClient method), 923
- receive_notification() (astropy.vo.samp.SAMPIntegratedClient method), 939
- receive_response() (astropy.vo.samp.SAMPClient method), 923
- receive_response() (astropy.vo.samp.SAMPIntegratedClient method), 940
- recommended_units (astropy.coordinates.BaseRepresentation attribute), 270
- recommended_units (astropy.coordinates.CylindricalRepresentation attribute), 275
- recommended_units (astropy.coordinates.PhysicsSphericalRepresentation attribute), 294
- recommended_units (astropy.coordinates.SphericalRepresentation attribute), 301
- recommended_units (astropy.coordinates.UnitSphericalRepresentation attribute), 308
- Redshift (class in astropy.modeling.functional_models), 437
- ref (astropy.io.votable.tree.Field attribute), 746
- ref (astropy.io.votable.tree.FieldRef attribute), 750
- ref (astropy.io.votable.tree.Group attribute), 752
- ref (astropy.io.votable.tree.Info attribute), 743
- ref (astropy.io.votable.tree.ParamRef attribute), 751
- ref (astropy.io.votable.tree.Table attribute), 754
- ref (astropy.io.votable.tree.Values attribute), 744
- reference (astropy.constants.Constant attribute), 22
- register() (astropy.coordinates.CoordinateTransform method), 274
- register() (astropy.units.NamedUnit method), 65
- register() (astropy.vo.samp.SAMPClient method), 924
- register() (astropy.vo.samp.SAMPHubProxy method), 927
- register_identifier() (in module astropy.io.registry), 961
- register_reader() (in module astropy.io.registry), 961
- register_writer() (in module astropy.io.registry), 961
- reject() (astropy.vo.samp.WebProfileDialog method), 942
- reload() (astropy.config.ConfigNamespace method), 957
- reload_config() (in module astropy.config), 954
- remote_timeout (astropy.utils.data.Conf attribute), 1007
- remove() (astropy.io.fits.CardList method), 633
- remove() (astropy.io.fits.Header method), 625
- remove_column() (astropy.table.Table method), 180
- remove_columns() (astropy.table.Table method), 181
- remove_row() (astropy.table.Table method), 181
- remove_rows() (astropy.table.Table method), 182
- remove_transform() (astropy.coordinates.TransformGraph method), 305
- rename_column() (astropy.table.Table method), 183
- rename_key() (astropy.io.fits.Header method), 625
- rename_keyword() (astropy.io.fits.Header method), 625
- reorient_celestial_first() (astropy.wcs.WCS method), 337
- replace_char (astropy.io.ascii.ContinuationLinesInputter attribute), 716
- replicate() (astropy.time.Time method), 205
- replicate() (astropy.time.TimeDelta method), 210
- reply() (astropy.vo.samp.SAMPHubProxy method), 927
- reply() (astropy.vo.samp.SAMPIntegratedClient method), 940
- report() (astropy.io.fits.FITSDiff method), 675
- report() (astropy.io.fits.HDUDiff method), 676

- report() (astropy.io.fits.HeaderDiff method), 678
- report() (astropy.io.fits.ImageDataDiff method), 679
- report() (astropy.io.fits.RawDataDiff method), 680
- report() (astropy.io.fits.TableDataDiff method), 681
- represent_as() (astropy.coordinates.BaseCoordinateFrame method), 268
- represent_as() (astropy.coordinates.BaseRepresentation method), 270
- represent_as() (astropy.coordinates.PhysicsSphericalRepresentation method), 294
- represent_as() (astropy.coordinates.SphericalRepresentation method), 302
- represent_as() (astropy.coordinates.UnitSphericalRepresentation method), 308
- representation (astropy.coordinates.BaseCoordinateFrame attribute), 266
- representation (astropy.coordinates.SkyCoord attribute), 297
- representation_component_names (astropy.coordinates.BaseCoordinateFrame attribute), 266
- representation_component_units (astropy.coordinates.BaseCoordinateFrame attribute), 267
- representation_info (astropy.coordinates.BaseCoordinateFrame attribute), 267
- RepresentationMapping (class in astropy.coordinates), 295
- req_cards() (astropy.io.fits.BinTableHDU method), 640
- req_cards() (astropy.io.fits.CompImageHDU method), 670
- req_cards() (astropy.io.fits.GroupsHDU method), 614
- req_cards() (astropy.io.fits.ImageHDU method), 658
- req_cards() (astropy.io.fits.PrimaryHDU method), 607
- req_cards() (astropy.io.fits.TableHDU method), 646
- reset() (astropy.config.ConfigNamespace method), 957
- Resource (class in astropy.io.votable.tree), 755
- resources (astropy.io.votable.tree.Resource attribute), 756
- resources (astropy.io.votable.tree.VOTableFile attribute), 758
- restfrq (astropy.wcs.Wcsprm attribute), 353
- restwav (astropy.wcs.Wcsprm attribute), 353
- results (astropy.utils.timer.RunTimePredictor attribute), 994
- reverse() (astropy.table.Table method), 183
- rho (astropy.coordinates.CylindricalRepresentation attribute), 275
- Ring2D (class in astropy.modeling.functional_models), 438
- Ring2DKernel (class in astropy.convolution), 811
- rotateCD() (astropy.wcs.WCS method), 338
- RotateCelestial2Native (class in astropy.modeling.rotations), 484
- RotateNative2Celestial (class in astropy.modeling.rotations), 484
- Rotation2D (class in astropy.modeling.rotations), 485
- Row (class in astropy.table), 169
- run_option() (astropy.io.fits.BinTableHDU method), 640
- run_option() (astropy.io.fits.Card method), 631
- run_option() (astropy.io.fits.CompImageHDU method), 671
- run_option() (astropy.io.fits.GroupsHDU method), 614
- run_option() (astropy.io.fits.ImageHDU method), 659
- run_option() (astropy.io.fits.PrimaryHDU method), 608
- run_option() (astropy.io.fits.TableHDU method), 646
- RunTimePredictor (class in astropy.utils.timer), 992
- ## S
- s2p() (astropy.wcs.Wcsprm method), 362
- SAMPClient (class in astropy.vo.samp), 918
- SAMPClientError, 925
- SAMPHubError, 925
- SAMPHubProxy (class in astropy.vo.samp), 925
- SAMPHubServer (class in astropy.vo.samp), 927
- SAMPIntegratedClient (class in astropy.vo.samp), 930
- SAMPMsgReplierWrapper (class in astropy.vo.samp), 941
- SAMPProxyError, 941
- SAMPWarning, 942
- save_config() (in module astropy.config), 955
- scale (astropy.time.Time attribute), 204
- scale (astropy.time.TimeFormat attribute), 213
- scale (astropy.units.CompositeUnit attribute), 62
- scale (astropy.units.UnitBase attribute), 78
- Scale (class in astropy.modeling.functional_models), 440
- scale() (astropy.io.fits.CompImageHDU method), 671
- scale() (astropy.io.fits.GroupsHDU method), 614
- scale() (astropy.io.fits.ImageHDU method), 659
- scale() (astropy.io.fits.PrimaryHDU method), 608
- scale_factor() (astropy.cosmology.FLRW method), 837
- scale_factor() (in module astropy.cosmology), 825
- SCALES (astropy.time.Time attribute), 203
- SCALES (astropy.time.TimeDelta attribute), 210
- ScaleValueError, 202
- ScienceState (class in astropy.utils.state), 996
- ScienceStateAlias (astropy.utils.state attribute), 997
- search_all() (in module astropy.vo.client.conesearch), 899
- searchsorted() (astropy.units.Quantity method), 73
- searchsorted() (astropy.units.quantity.Quantity method), 51
- section (astropy.io.fits.GroupsHDU attribute), 615
- section (astropy.io.fits.ImageHDU attribute), 660
- section (astropy.io.fits.PrimaryHDU attribute), 608
- Section (class in astropy.io.fits), 674
- sense (astropy.wcs.Tabprm attribute), 328
- separable (astropy.convolution.Kernel attribute), 804

- separation() (astropy.coordinates.BaseCoordinateFrame method), 268
- separation() (astropy.coordinates.SkyCoord method), 299
- separation_3d() (astropy.coordinates.BaseCoordinateFrame method), 269
- separation_3d() (astropy.coordinates.SkyCoord method), 300
- SerialCompositeModel (class in astropy.modeling), 399
- set() (astropy.io.fits.Header method), 626
- set() (astropy.utils.state.ScienceState class method), 997
- set() (astropy.wcs.Tabprm method), 328
- set() (astropy.wcs.Wcsprm method), 363
- set_all_tables_format() (astropy.io.votable.tree.VOTableFile method), 760
- set_current() (in module astropy.cosmology), 825
- set_enabled_equivalencies() (in module astropy.units), 60
- set_enabled_units() (in module astropy.units), 60
- set_guess() (in module astropy.io.ascii), 703
- set_jds() (astropy.time.TimeDatetime method), 209
- set_jds() (astropy.time.TimeDeltaFormat method), 210
- set_jds() (astropy.time.TimeEpochDate method), 212
- set_jds() (astropy.time.TimeEpochDateString method), 212
- set_jds() (astropy.time.TimeFormat method), 213
- set_jds() (astropy.time.TimeFromEpoch method), 214
- set_jds() (astropy.time.TimeJD method), 217
- set_jds() (astropy.time.TimeMJD method), 219
- set_jds() (astropy.time.TimeString method), 220
- set_ps() (astropy.wcs.Wcsprm method), 363
- set_pv() (astropy.wcs.Wcsprm method), 363
- set_temp() (astropy.config.ConfigNamespace method), 957
- set_xmlrpc_callback() (astropy.vo.samp.SAMPHubProxy method), 927
- setdefault() (astropy.io.fits.Header method), 627
- setfield() (astropy.io.fits.FITS_record method), 652
- setLevel() (astropy.logger.AstropyLogger method), 970
- setpar() (astropy.io.fits.Group method), 618
- setter() (astropy.utils.misc.lazyproperty method), 981
- setval() (in module astropy.io.fits), 599
- SExtractor (class in astropy.io.ascii), 729
- shape (astropy.convolution.Kernel attribute), 804
- shape (astropy.coordinates.BaseCoordinateFrame attribute), 267
- shape (astropy.coordinates.BaseRepresentation attribute), 270
- shape (astropy.io.fits.CompImageHDU attribute), 671
- shape (astropy.io.fits.GroupsHDU attribute), 615
- shape (astropy.io.fits.ImageHDU attribute), 660
- shape (astropy.io.fits.PrimaryHDU attribute), 608
- shape (astropy.modeling.Parameter attribute), 398
- shape (astropy.nddata.NDData attribute), 108
- Shift (class in astropy.modeling.functional_models), 440
- short_names (astropy.units.NamedUnit attribute), 65
- show_in_browser() (astropy.table.Table method), 184
- si (astropy.constants.Constant attribute), 22
- si (astropy.units.Quantity attribute), 68
- si (astropy.units.quantity.Quantity attribute), 46
- si (astropy.units.UnitBase attribute), 78
- sidereal_time() (astropy.time.Time method), 206
- sigma (astropy.modeling.functional_models.MexicanHat1D attribute), 435
- sigma (astropy.modeling.functional_models.MexicanHat2D attribute), 437
- sigma_clip() (in module astropy.stats), 862
- signal_to_noise_oir_ccd() (in module astropy.stats), 864
- signed_dms (astropy.coordinates.Angle attribute), 263
- silence() (in module astropy.utils.misc), 978
- Simplex (class in astropy.modeling.optimizers), 450
- SimplexLSQFitter (class in astropy.modeling.fitting), 405
- Sine1D (class in astropy.modeling.functional_models), 441
- SingularMatrixError, 323
- sip (astropy.wcs.WCSBase attribute), 345
- SIP (class in astropy.modeling.polynomial), 468
- Sip (class in astropy.wcs), 324
- sip_foc2pix() (astropy.wcs.WCS method), 338
- sip_pix2foc() (astropy.wcs.WCS method), 338
- size (astropy.io.fits.BinTableHDU attribute), 640
- size (astropy.io.fits.CompImageHDU attribute), 671
- size (astropy.io.fits.GroupsHDU attribute), 615
- size (astropy.io.fits.ImageHDU attribute), 660
- size (astropy.io.fits.PrimaryHDU attribute), 608
- size (astropy.io.fits.StreamingHDU attribute), 618
- size (astropy.io.fits.TableHDU attribute), 646
- size (astropy.modeling.Parameter attribute), 398
- size (astropy.nddata.NDData attribute), 108
- skipinitialspace (astropy.io.ascii.DefaultSplitter attribute), 719
- Sky2Pix_AZP (class in astropy.modeling.projections), 473
- Sky2Pix_CAR (class in astropy.modeling.projections), 475
- Sky2Pix_CEA (class in astropy.modeling.projections), 476
- Sky2Pix_CYP (class in astropy.modeling.projections), 477
- Sky2Pix_MER (class in astropy.modeling.projections), 478
- Sky2Pix_SIN (class in astropy.modeling.projections), 479
- Sky2Pix_STG (class in astropy.modeling.projections), 480
- Sky2Pix_TAN (class in astropy.modeling.projections), 481
- SkyCoord (class in astropy.coordinates), 295
- slice() (astropy.wcs.WCS method), 339

- slope (astropy.modeling.functional_models.Linear1D attribute), 432
- slope (astropy.modeling.functional_models.Trapezoid1D attribute), 444
- slope (astropy.modeling.functional_models.TrapezoidDisk2D attribute), 447
- SLSQP (class in astropy.modeling.optimizers), 449
- SLSQPLSQFitter (class in astropy.modeling.fitting), 404
- sort() (astropy.table.Table method), 184
- spcfix() (astropy.wcs.Wcsprm method), 364
- spec (astropy.wcs.Wcsprm attribute), 353
- specsys (astropy.wcs.Wcsprm attribute), 353
- spectral() (in module astropy.units), 61
- spectral() (in module astropy.units.equivalencies), 95
- spectral_density() (in module astropy.units), 61
- spectral_density() (in module astropy.units.equivalencies), 95
- spherical (astropy.coordinates.BaseCoordinateFrame attribute), 267
- spherical_to_cartesian() (in module astropy.coordinates), 259
- SphericalRepresentation (class in astropy.coordinates), 301
- Spinner (class in astropy.utils.console), 990
- sptr() (astropy.wcs.Wcsprm method), 364
- ssysobs (astropy.wcs.Wcsprm attribute), 353
- ssyssrc (astropy.wcs.Wcsprm attribute), 354
- standard_broadcasting (astropy.modeling.Model attribute), 396
- standard_broadcasting (astropy.modeling.projections.AffineTransformation2D attribute), 482
- start() (astropy.utils.xml.writer.XMLWriter method), 1012
- start() (astropy.vo.samp.SAMPClient method), 924
- start() (astropy.vo.samp.SAMPHubServer method), 930
- start_line (astropy.io.ascii.BaseData attribute), 706
- start_line (astropy.io.ascii.BaseHeader attribute), 708
- StaticMatrixTransform (class in astropy.coordinates), 302
- std() (astropy.units.Quantity method), 73
- std() (astropy.units.quantity.Quantity method), 51
- stddev (astropy.modeling.functional_models.Gaussian1D attribute), 425
- stddev (astropy.modeling.functional_models.GaussianAbsorption1D attribute), 430
- StdDevUncertainty (class in astropy.nddata), 113
- stop() (astropy.vo.samp.SAMPClient method), 924
- stop() (astropy.vo.samp.SAMPHubServer method), 930
- str_kwargs() (astropy.time.TimeString method), 220
- StreamingHDU (class in astropy.io.fits), 618
- strict_names (astropy.io.ascii.BaseReader attribute), 710
- StrType (class in astropy.io.ascii), 730
- sub() (astropy.wcs.WCS method), 339
- sub() (astropy.wcs.Wcsprm method), 364
- subfmts (astropy.time.TimeISO attribute), 215
- subfmts (astropy.time.TimeISOT attribute), 216
- subfmts (astropy.time.TimeYearDayTime attribute), 221
- subtract() (astropy.nddata.NDData method), 110
- sum() (astropy.units.Quantity method), 74
- sum() (astropy.units.quantity.Quantity method), 52
- SummedCompositeModel (class in astropy.modeling), 400
- support_correlated (astropy.nddata.StdDevUncertainty attribute), 113
- supported_constraints (astropy.modeling.fitting.LevMarLSQFitter attribute), 403
- supported_constraints (astropy.modeling.fitting.LinearLSQFitter attribute), 401
- supported_constraints (astropy.modeling.fitting.SimplexLSQFitter attribute), 405
- supported_constraints (astropy.modeling.fitting.SLSQPLSQFitter attribute), 404
- supported_constraints (astropy.modeling.optimizers.Optimization attribute), 448
- supported_constraints (astropy.modeling.optimizers.Simplex attribute), 450
- supported_constraints (astropy.modeling.optimizers.SLSQP attribute), 449
- supports_correlated (astropy.nddata.NDUncertainty attribute), 111
- supports_empty_values() (astropy.io.votable.converters.Converter method), 763
- swapaxes() (astropy.wcs.WCS method), 340
- system (astropy.constants.Constant attribute), 22
- system (astropy.io.votable.tree.CooSys attribute), 750

T

- tab (astropy.wcs.Wcsprm attribute), 354
- Tab (class in astropy.io.ascii), 730
- table (astropy.table.Row attribute), 170
- Table (class in astropy.io.votable.tree), 753
- Table (class in astropy.table), 170
- table_column_to_votable_datatype() (in module astropy.io.votable.converters), 761
- TableColumns (class in astropy.table), 186
- TableDataDiff (class in astropy.io.fits), 680
- tabledump() (in module astropy.io.fits), 653
- TableFormatter (class in astropy.table), 187
- TableGroups (class in astropy.table), 187
- TableHDU (class in astropy.io.fits), 643

- [tableload\(\)](#) (in module `astropy.io.fits`), 654
[TableMergeError](#), 188
[TableOutputter](#) (class in `astropy.io.ascii`), 731
[tables](#) (`astropy.io.votable.tree.Resource` attribute), 756
[Tabprm](#) (class in `astropy.wcs`), 326
[tag\(\)](#) (`astropy.utils.xml.writer.XMLWriter` method), 1013
[tally\(\)](#) (`astropy.vo.validator.inspect.ConeSearchResults` method), 907
[Tcmb\(\)](#) (`astropy.cosmology.FLRW` method), 831
[Tcmb0](#) (`astropy.cosmology.FLRW` attribute), 829
[tcreate\(\)](#) (`astropy.io.fits.BinTableHDU` class method), 640
[tcreate\(\)](#) (`astropy.io.fits.CompImageHDU` class method), 671
[tdump\(\)](#) (`astropy.io.fits.BinTableHDU` method), 641
[tdump\(\)](#) (`astropy.io.fits.CompImageHDU` method), 671
[temperature\(\)](#) (in module `astropy.units`), 61
[temperature\(\)](#) (in module `astropy.units.equivalencies`), 99
[temperature_energy\(\)](#) (in module `astropy.units`), 61
[temperature_energy\(\)](#) (in module `astropy.units.equivalencies`), 99
[terminal_size\(\)](#) (in module `astropy.utils.console`), 988
[theta](#) (`astropy.coordinates.PhysicsSphericalRepresentation` attribute), 294
[theta](#) (`astropy.modeling.functional_models.Gaussian2D` attribute), 428
[theta0](#) (`astropy.wcs.Wcsprm` attribute), 354
[tied](#) (`astropy.modeling.Model` attribute), 396
[tied](#) (`astropy.modeling.Parameter` attribute), 398
[Time](#) (class in `astropy.time`), 202
[time_func\(\)](#) (`astropy.utils.timer.RunTimePredictor` method), 996
[TimeBesselianEpoch](#) (class in `astropy.time`), 206
[TimeBesselianEpochString](#) (class in `astropy.time`), 207
[TimeCxcSec](#) (class in `astropy.time`), 207
[TimeDatetime](#) (class in `astropy.time`), 208
[TimeDelta](#) (class in `astropy.time`), 209
[TimeDeltaFormat](#) (class in `astropy.time`), 210
[TimeDeltaJD](#) (class in `astropy.time`), 211
[TimeDeltaSec](#) (class in `astropy.time`), 211
[TimeEpochDate](#) (class in `astropy.time`), 211
[TimeEpochDateString](#) (class in `astropy.time`), 212
[TimeFormat](#) (class in `astropy.time`), 213
[TimeFrameAttribute](#) (class in `astropy.coordinates`), 303
[TimeFromEpoch](#) (class in `astropy.time`), 214
[timefunc\(\)](#) (in module `astropy.utils.timer`), 991
[TimeGPS](#) (class in `astropy.time`), 214
[TimeISO](#) (class in `astropy.time`), 215
[TimeISOT](#) (class in `astropy.time`), 216
[TimeJD](#) (class in `astropy.time`), 216
[TimeJulianEpoch](#) (class in `astropy.time`), 217
[TimeJulianEpochString](#) (class in `astropy.time`), 217
[TimeMJD](#) (class in `astropy.time`), 218
[TimePlotDate](#) (class in `astropy.time`), 219
[TimeString](#) (class in `astropy.time`), 220
[TimeUnix](#) (class in `astropy.time`), 220
[TimeYearDayTime](#) (class in `astropy.time`), 221
[Tnu\(\)](#) (`astropy.cosmology.FLRW` method), 831
[Tnu0](#) (`astropy.cosmology.FLRW` attribute), 830
[to\(\)](#) (`astropy.coordinates.EarthLocation` method), 282
[to\(\)](#) (`astropy.time.TimeDelta` method), 210
[to\(\)](#) (`astropy.units.Quantity` method), 74
[to\(\)](#) (`astropy.units.quantity.Quantity` method), 52
[to\(\)](#) (`astropy.units.UnitBase` method), 80
[to_cartesian\(\)](#) (`astropy.coordinates.BaseRepresentation` method), 270
[to_cartesian\(\)](#) (`astropy.coordinates.CartesianRepresentation` method), 272
[to_cartesian\(\)](#) (`astropy.coordinates.CylindricalRepresentation` method), 276
[to_cartesian\(\)](#) (`astropy.coordinates.PhysicsSphericalRepresentation` method), 294
[to_cartesian\(\)](#) (`astropy.coordinates.SphericalRepresentation` method), 302
[to_cartesian\(\)](#) (`astropy.coordinates.UnitSphericalRepresentation` method), 308
[to_dot_graph\(\)](#) (`astropy.coordinates.TransformGraph` method), 306
[to_fits\(\)](#) (`astropy.wcs.WCS` method), 341
[to_geocentric\(\)](#) (`astropy.coordinates.EarthLocation` method), 282
[to_geodetic\(\)](#) (`astropy.coordinates.EarthLocation` method), 282
[to_header\(\)](#) (`astropy.wcs.WCS` method), 341
[to_header\(\)](#) (`astropy.wcs.Wcsprm` method), 366
[to_header_string\(\)](#) (`astropy.wcs.WCS` method), 342
[to_json\(\)](#) (`astropy.vo.client.vos_catalog.VOSDatabase` method), 897
[to_networkx_graph\(\)](#) (`astropy.coordinates.TransformGraph` method), 306
[to_string\(\)](#) (`astropy.coordinates.Angle` method), 263
[to_string\(\)](#) (`astropy.coordinates.SkyCoord` method), 300
[to_string\(\)](#) (`astropy.units.format.Base` method), 84
[to_string\(\)](#) (`astropy.units.format.CDS` method), 85
[to_string\(\)](#) (`astropy.units.format.Console` method), 86
[to_string\(\)](#) (`astropy.units.format.Fits` method), 86
[to_string\(\)](#) (`astropy.units.format.Generic` method), 85
[to_string\(\)](#) (`astropy.units.format.Latex` method), 87
[to_string\(\)](#) (`astropy.units.format.OGIP` method), 87
[to_string\(\)](#) (`astropy.units.format.VOUnit` method), 88
[to_string\(\)](#) (`astropy.units.UnitBase` method), 81
[to_string\(\)](#) (`astropy.units.UnrecognizedUnit` method), 83
[to_system\(\)](#) (`astropy.units.UnitBase` method), 81
[to_table\(\)](#) (`astropy.io.votable.tree.Table` method), 755
[to_table_column\(\)](#) (`astropy.io.votable.tree.Field` method), 747
[to_table_column\(\)](#) (`astropy.io.votable.tree.Link` method), 742

- to_table_column() (astropy.io.votable.tree.Values method), 745
- to_xml() (astropy.io.votable.tree.Field method), 747
- to_xml() (astropy.io.votable.tree.Group method), 752
- to_xml() (astropy.io.votable.tree.Info method), 743
- to_xml() (astropy.io.votable.tree.Param method), 748
- to_xml() (astropy.io.votable.tree.Resource method), 757
- to_xml() (astropy.io.votable.tree.Table method), 755
- to_xml() (astropy.io.votable.tree.Values method), 745
- to_xml() (astropy.io.votable.tree.VOTableFile method), 760
- tofile() (astropy.io.fits.Header method), 627
- tofile() (astropy.units.Quantity method), 74
- tofile() (astropy.units.quantity.Quantity method), 52
- Tophat2DKernel (class in astropy.convolution), 812
- tostring() (astropy.io.fits.Header method), 627
- tostring() (astropy.units.Quantity method), 74
- tostring() (astropy.units.quantity.Quantity method), 52
- totextfile() (astropy.io.fits.Header method), 628
- toTxtFile() (astropy.io.fits.Header method), 627
- trace() (astropy.units.Quantity method), 75
- trace() (astropy.units.quantity.Quantity method), 53
- transform() (astropy.coordinates.TransformGraph method), 306
- transform_to() (astropy.coordinates.BaseCoordinateFrame method), 269
- transform_to() (astropy.coordinates.SkyCoord method), 300
- TransformGraph (class in astropy.coordinates), 303
- translation (astropy.modeling.projections.AffineTransform attribute), 482
- Trapezoid1D (class in astropy.modeling.functional_models), 443
- Trapezoid1DKernel (class in astropy.convolution), 812
- TrapezoidDisk2D (class in astropy.modeling.functional_models), 445
- TrapezoidDisk2DKernel (class in astropy.convolution), 813
- truncation (astropy.convolution.Kernel attribute), 804
- type (astropy.io.votable.tree.Field attribute), 746
- type (astropy.io.votable.tree.Resource attribute), 756
- type (astropy.io.votable.tree.Values attribute), 745
- ## U
- unbind_receive_call() (astropy.vo.samp.SAMPClient method), 924
- unbind_receive_call() (astropy.vo.samp.SAMPIntegratedClient method), 940
- unbind_receive_notification() (astropy.vo.samp.SAMPClient method), 924
- unbind_receive_notification() (astropy.vo.samp.SAMPIntegratedClient method), 941
- unbind_receive_response() (astropy.vo.samp.SAMPClient method), 924
- unbind_receive_response() (astropy.vo.samp.SAMPIntegratedClient method), 941
- uncertainty (astropy.constants.Constant attribute), 22
- uncertainty (astropy.nddata.NDData attribute), 108
- unescape_all() (in module astropy.utils.xml.unescaper), 1010
- Unicode (class in astropy.units.format), 87
- UnimplementedWarning (class in astropy.io.votable.exceptions), 779
- uniqify_names() (astropy.io.votable.tree.Field class method), 747
- unit (astropy.constants.Constant attribute), 22
- unit (astropy.io.votable.tree.Field attribute), 747
- unit (astropy.io.votable.tree.Info attribute), 743
- unit (astropy.nddata.NDData attribute), 108
- unit (astropy.time.TimeCxcSec attribute), 208
- unit (astropy.time.TimeDeltaJD attribute), 211
- unit (astropy.time.TimeDeltaSec attribute), 211
- unit (astropy.time.TimeGPS attribute), 215
- unit (astropy.time.TimeJulianEpoch attribute), 217
- unit (astropy.time.TimePlotDate attribute), 219
- unit (astropy.time.TimeUnix attribute), 221
- unit (astropy.units.Quantity attribute), 68
- unit (astropy.units.quantity.Quantity attribute), 46
- Unit (class in astropy.units), 75
- UnitBase (class in astropy.units), 77
- unitID() (astropy.wcs.Wcsprm method), 367
- UnitsError, 81
- UnitSphericalRepresentation (class in astropy.coordinates), 307
- UnitsWarning, 81
- UnrecognizedUnit (class in astropy.units), 81
- unregister() (astropy.coordinates.CoordinateTransform method), 274
- unregister() (astropy.vo.samp.SAMPClient method), 924
- unregister() (astropy.vo.samp.SAMPHubProxy method), 927
- Unscaled (class in astropy.units.format), 87
- update() (astropy.io.fits.BinTableHDU method), 641
- update() (astropy.io.fits.CompImageHDU method), 671
- update() (astropy.io.fits.Header method), 628
- update() (astropy.io.fits.TableHDU method), 646
- update() (astropy.utils.console.ProgressBar method), 989
- update() (astropy.utils.console.ProgressBarOrSpinner method), 991
- update() (in module astropy.io.fits), 596
- update_ext_name() (astropy.io.fits.BinTableHDU method), 641
- update_ext_name() (astropy.io.fits.CompImageHDU method), 672

- update_ext_name() (astropy.io.fits.GroupsHDU method), 615
 - update_ext_name() (astropy.io.fits.ImageHDU method), 660
 - update_ext_name() (astropy.io.fits.PrimaryHDU method), 608
 - update_ext_name() (astropy.io.fits.TableHDU method), 646
 - update_ext_version() (astropy.io.fits.BinTableHDU method), 641
 - update_ext_version() (astropy.io.fits.CompImageHDU method), 672
 - update_ext_version() (astropy.io.fits.GroupsHDU method), 615
 - update_ext_version() (astropy.io.fits.ImageHDU method), 660
 - update_ext_version() (astropy.io.fits.PrimaryHDU method), 609
 - update_ext_version() (astropy.io.fits.TableHDU method), 647
 - update_extend() (astropy.io.fits.HDUList method), 603
 - update_header() (astropy.io.fits.ImageHDU method), 661
 - update_meta() (astropy.io.ascii.BaseHeader method), 708
 - updateCompressedData() (astropy.io.fits.CompImageHDU method), 671
 - updateHeader() (astropy.io.fits.CompImageHDU method), 671
 - updateHeaderData() (astropy.io.fits.CompImageHDU method), 672
 - upper_key() (in module astropy.io.fits), 634
 - use_internet (astropy.vo.samp.Conf attribute), 918
- ## V
- val (astropy.time.Time attribute), 204
 - validate() (astropy.cosmology.default_cosmology class method), 845
 - validate() (astropy.utils.state.ScienceState class method), 997
 - validate() (in module astropy.io.votable), 740
 - validate() (in module astropy.wcs), 321
 - validate_schema() (in module astropy.io.votable.xmlutil), 767
 - validate_schema() (in module astropy.utils.xml.validate), 1010
 - ValidationMultiprocessingError, 908
 - vals (astropy.time.Time attribute), 204
 - value (astropy.io.fits.Card attribute), 631
 - value (astropy.io.votable.tree.Info attribute), 743
 - value (astropy.io.votable.tree.Param attribute), 748
 - value (astropy.modeling.Parameter attribute), 398
 - value (astropy.time.Time attribute), 204
 - value (astropy.time.TimeDatetime attribute), 209
 - value (astropy.time.TimeDeltaFormat attribute), 210
 - value (astropy.time.TimeEpochDate attribute), 212
 - value (astropy.time.TimeEpochDateString attribute), 212
 - value (astropy.time.TimeFormat attribute), 213
 - value (astropy.time.TimeFromEpoch attribute), 214
 - value (astropy.time.TimeJD attribute), 217
 - value (astropy.time.TimeMJD attribute), 218
 - value (astropy.time.TimeString attribute), 220
 - value (astropy.units.Quantity attribute), 68
 - value (astropy.units.quantity.Quantity attribute), 46
 - values (astropy.io.votable.tree.Field attribute), 747
 - Values (class in astropy.io.votable.tree), 744
 - values() (astropy.io.fits.CardList method), 633
 - values() (astropy.io.fits.Header method), 630
 - values() (astropy.table.TableColumns method), 186
 - var() (astropy.units.Quantity method), 75
 - var() (astropy.units.quantity.Quantity method), 53
 - velangl (astropy.wcs.Wcsprm attribute), 354
 - velosys (astropy.wcs.Wcsprm attribute), 354
 - verify() (astropy.io.fits.BinTableHDU method), 642
 - verify() (astropy.io.fits.Card method), 631
 - verify() (astropy.io.fits.CompImageHDU method), 673
 - verify() (astropy.io.fits.GroupsHDU method), 616
 - verify() (astropy.io.fits.ImageHDU method), 661
 - verify() (astropy.io.fits.PrimaryHDU method), 609
 - verify() (astropy.io.fits.TableHDU method), 648
 - verify_checksum() (astropy.io.fits.BinTableHDU method), 642
 - verify_checksum() (astropy.io.fits.CompImageHDU method), 673
 - verify_checksum() (astropy.io.fits.GroupsHDU method), 616
 - verify_checksum() (astropy.io.fits.ImageHDU method), 661
 - verify_checksum() (astropy.io.fits.PrimaryHDU method), 609
 - verify_checksum() (astropy.io.fits.TableHDU method), 648
 - verify_datasum() (astropy.io.fits.BinTableHDU method), 642
 - verify_datasum() (astropy.io.fits.CompImageHDU method), 673
 - verify_datasum() (astropy.io.fits.GroupsHDU method), 616
 - verify_datasum() (astropy.io.fits.ImageHDU method), 661
 - verify_datasum() (astropy.io.fits.PrimaryHDU method), 610
 - verify_datasum() (astropy.io.fits.TableHDU method), 648
 - version (astropy.io.votable.tree.VOTableFile attribute), 758
 - version (astropy.vo.client.vos_catalog.VOSDatabase attribute), 894
 - vo_raise() (in module astropy.io.votable.exceptions), 779
 - vo_reraise() (in module astropy.io.votable.exceptions), 779

- vo_warn() (in module astropy.io.votable.exceptions), 779
 vos_baseurl (astropy.vo.Conf attribute), 889
 VOSBase (class in astropy.vo.client.vos_catalog), 891
 VOSCatalog (class in astropy.vo.client.vos_catalog), 892
 VOSDatabase (class in astropy.vo.client.vos_catalog), 893
 VOSError, 904
 VOTableChangeWarning (class in astropy.io.votable.exceptions), 779
 VOTableFile (class in astropy.io.votable.tree), 757
 VOTableSpecError (class in astropy.io.votable.exceptions), 779
 VOTableSpecWarning (class in astropy.io.votable.exceptions), 779
 VOUnit (class in astropy.units.format), 88
 VOWarning (class in astropy.io.votable.exceptions), 779
 vstack() (in module astropy.table), 158
- ## W
- w() (astropy.cosmology.FLRW method), 838
 w() (astropy.cosmology.LambdaCDM method), 844
 w() (astropy.cosmology.w0waCDM method), 847
 w() (astropy.cosmology.w0wzCDM method), 849
 w() (astropy.cosmology.wCDM method), 851
 w() (astropy.cosmology.wpwaCDM method), 854
 w0 (astropy.cosmology.w0waCDM attribute), 846
 w0 (astropy.cosmology.w0wzCDM attribute), 848
 w0 (astropy.cosmology.wCDM attribute), 850
 w0waCDM (class in astropy.cosmology), 845
 w0wzCDM (class in astropy.cosmology), 847
 wa (astropy.cosmology.w0waCDM attribute), 846
 wa (astropy.cosmology.wpwaCDM attribute), 853
 walk_skip_hidden() (in module astropy.utils.misc), 980
 warn_or_raise() (in module astropy.io.votable.exceptions), 779
 warn_setting_unit_directly (astropy.nddata.Conf attribute), 105
 warn_unsupported_correlated (astropy.nddata.Conf attribute), 105
 warnings_logging_enabled() (astropy.logger.AstropyLogger method), 970
 wCDM (class in astropy.cosmology), 849
 wcs (astropy.wcs.WCSBase attribute), 345
 WCS (class in astropy.wcs), 328
 wcs_pix2world() (astropy.wcs.WCS method), 342
 wcs_world2pix() (astropy.wcs.WCS method), 343
 WCSBase (class in astropy.wcs), 344
 WcsError, 345
 Wcsprm (class in astropy.wcs), 345
 WebProfileDialog (class in astropy.vo.samp), 942
 WhitespaceSplitter (class in astropy.io.ascii), 731
 width (astropy.io.votable.tree.Field attribute), 747
 width (astropy.modeling.functional_models.Box1D attribute), 416
 width (astropy.modeling.functional_models.Ring2D attribute), 439
 width (astropy.modeling.functional_models.Trapezoid1D attribute), 444
 wp (astropy.cosmology.wpwaCDM attribute), 853
 wpwaCDM (class in astropy.cosmology), 852
 wrap_angle (astropy.coordinates.Longitude attribute), 293
 wrap_at() (astropy.coordinates.Angle method), 265
 write() (astropy.io.ascii.BaseData method), 707
 write() (astropy.io.ascii.BaseHeader method), 708
 write() (astropy.io.ascii.BaseReader method), 711
 write() (astropy.io.ascii.Cds method), 714
 write() (astropy.io.ascii.Daophot method), 718
 write() (astropy.io.ascii.FixedWidthData method), 720
 write() (astropy.io.ascii.FixedWidthHeader method), 722
 write() (astropy.io.ascii.HTML method), 725
 write() (astropy.io.ascii.Ipac method), 727
 write() (astropy.io.ascii.Latex method), 728
 write() (astropy.io.ascii.SExtractor method), 730
 write() (astropy.io.fits.StreamingHDU method), 619
 write() (astropy.nddata.NDData method), 111
 write() (astropy.table.Table method), 185
 write() (in module astropy.io.ascii), 704
 write() (in module astropy.io.registry), 963
 write_spacer_lines (astropy.io.ascii.BaseData attribute), 706
 write_spacer_lines (astropy.io.ascii.BaseHeader attribute), 708
 write_table_hdf5() (in module astropy.io.misc.hdf5), 783
 writeto() (astropy.io.fits.BinTableHDU method), 642
 writeto() (astropy.io.fits.CompImageHDU method), 673
 writeto() (astropy.io.fits.GroupsHDU method), 617
 writeto() (astropy.io.fits.HDUList method), 603
 writeto() (astropy.io.fits.ImageHDU method), 662
 writeto() (astropy.io.fits.PrimaryHDU method), 610
 writeto() (astropy.io.fits.TableHDU method), 648
 writeto() (in module astropy.io.fits), 595
 writeto() (in module astropy.io.votable), 741
 wz (astropy.cosmology.w0wzCDM attribute), 848
- ## X
- x (astropy.coordinates.CartesianRepresentation attribute), 272
 x (astropy.coordinates.EarthLocation attribute), 281
 x_0 (astropy.modeling.functional_models.AiryDisk2D attribute), 411
 x_0 (astropy.modeling.functional_models.Beta1D attribute), 413
 x_0 (astropy.modeling.functional_models.Beta2D attribute), 415
 x_0 (astropy.modeling.functional_models.Box1D attribute), 416

[x_0 \(astropy.modeling.functional_models.Box2D attribute\), 418](#)
[x_0 \(astropy.modeling.functional_models.Disk2D attribute\), 423](#)
[x_0 \(astropy.modeling.functional_models.Lorentz1D attribute\), 433](#)
[x_0 \(astropy.modeling.functional_models.MexicanHat1D attribute\), 435](#)
[x_0 \(astropy.modeling.functional_models.MexicanHat2D attribute\), 437](#)
[x_0 \(astropy.modeling.functional_models.Ring2D attribute\), 439](#)
[x_0 \(astropy.modeling.functional_models.Trapezoid1D attribute\), 444](#)
[x_0 \(astropy.modeling.functional_models.TrapezoidDisk2D attribute\), 447](#)
[x_0 \(astropy.modeling.powerlaws.ExponentialCutoffPowerLaw1D attribute\), 453](#)
[x_0 \(astropy.modeling.powerlaws.LogParabola1D attribute\), 455](#)
[x_0 \(astropy.modeling.powerlaws.PowerLaw1D attribute\), 456](#)
[x_break \(astropy.modeling.powerlaws.BrokenPowerLaw1D attribute\), 452](#)
[x_cutoff \(astropy.modeling.powerlaws.ExponentialCutoffPowerLaw1D attribute\), 453](#)
[x_mean \(astropy.modeling.functional_models.Gaussian2D attribute\), 428](#)
[x_stddev \(astropy.modeling.functional_models.Gaussian2D attribute\), 428](#)
[x_width \(astropy.modeling.functional_models.Box2D attribute\), 418](#)
[xml_readlines\(\) \(in module astropy.utils.xml.iterparser\), 1009](#)
[XMLWriter \(class in astropy.utils.xml.writer\), 1011](#)
[xyz \(astropy.coordinates.CartesianRepresentation attribute\), 272](#)

Y

[y \(astropy.coordinates.CartesianRepresentation attribute\), 272](#)
[y \(astropy.coordinates.EarthLocation attribute\), 281](#)
[y_0 \(astropy.modeling.functional_models.AiryDisk2D attribute\), 411](#)
[y_0 \(astropy.modeling.functional_models.Beta2D attribute\), 415](#)
[y_0 \(astropy.modeling.functional_models.Box2D attribute\), 418](#)
[y_0 \(astropy.modeling.functional_models.Disk2D attribute\), 423](#)
[y_0 \(astropy.modeling.functional_models.MexicanHat2D attribute\), 437](#)
[y_0 \(astropy.modeling.functional_models.Ring2D attribute\), 439](#)

Z

[z \(astropy.coordinates.CartesianRepresentation attribute\), 272](#)
[z \(astropy.coordinates.CylindricalRepresentation attribute\), 275](#)
[z \(astropy.coordinates.Distance attribute\), 277](#)
[z \(astropy.coordinates.EarthLocation attribute\), 281](#)
[z \(astropy.modeling.functional_models.Redshift attribute\), 438](#)
[z_at_value\(\) \(in module astropy.cosmology\), 826](#)
[zp \(astropy.cosmology.wpwaCDM attribute\), 853](#)
[zsource \(astropy.wcs.Wcsprm attribute\), 354](#)